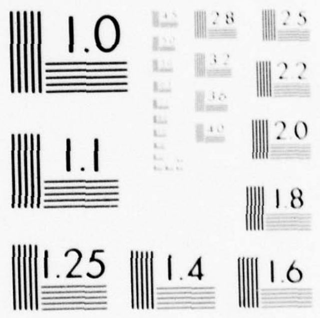
AD-A072 395 OPERATING SYSTEMS INC WOODLAND CA F/G 9/2
A KNOWLEDGE-BASED AUTOMATED MESSAGE UNDERSTANDING METHODOLOGY F--ETC(U)
JUN 79 G M SILVA, D L DWIGGINS, S G BUSBY F30602-77-C-0144
UNCLASSIFIED RADC-TR-79-133 NL

1 OF 2
AD A072395


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

12

RADC-TR-79-133
Final Technical Report
June 1979

A KNOWLEDGE-BASED AUTOMATED MESSAGE UNDERSTANDING METHODOLOGY FOR AN ADVANCED INDICATIONS SYSTEM

Operating Systems, Inc.

G.M.T. Silva
D.L. Dwiggins
S.G. Busby
J.L. Kuhns

LEVEL II

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

A KNOWLEDGE-BASED AUTOMATED MESSAGE UNDERSTANDING METHODOLOGY FOR AN ADVANCED INDICATIONS SYSTEM

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

79 08 06 014

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-133 has been reviewed and is approved for publication.

APPROVED: [illegible]

ANDREW S. KOZAK
Project Engineer

APPROVED: [illegible]

HOWARD DAVIS
Technical Director
Intelligence and Reconnaissance Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
(18) 1. REPORT NUMBER RADC-TR-79-133	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER (9)
(6) 4. TITLE (and Subtitle) A KNOWLEDGE-BASED AUTOMATED MESSAGE UNDERSTANDING METHODOLOGY FOR AN ADVANCED INDICATIONS SYSTEM.		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report. 12 Jul 77 — 17 Jan 79
		6. PERFORMING ORG. REPORT NUMBER N/A
(10) 7. AUTHOR(s) G.M.T. Silva, J.L. Kuhns, D.L. Dwiggins, S.G. Busby		8. CONTRACT OR GRANT NUMBER(s) (15) F30602-77-C-0144 new OK
9. PERFORMING ORGANIZATION NAME AND ADDRESS Operating Systems, Inc. 21031 Ventura Boulevard Woodland Hills CA 91364		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F (16) 45941238 (17) 12
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (IRDE) Griffiss AFB NY 13441 OK		(11) 12. REPORT DATE June [redacted]79
		13. NUMBER OF PAGES 193
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same (12) 192 p.		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Andrew S. Kozak (IRDE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Intelligence Data Processing
Data Base Generation
Natural Language Processing
Artificial Intelligence
Computational lnguistics
Logic Programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes an RADC sponsored effort related to the development of a technology based upon a computer "understanding" of natural language, with the aim of deriving fixed-format problem-oriented information records from the narrative text of intelligence messages in support of I&W functions.

The introductory section provides an overview of the concepts that serve as a foundation for the RADC developmental program for both interactive and automated exploitation of the content of intelligence messages, summarizes the

Over

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

391 994 Du

Item 20 (Cont'd)

work performed under the current effort, and presents our general conclusions.

Section 2 describes some of the fundamental concepts underlying message text analysis. It discusses the structure of event reports as viewed from four different perspectives; develops the concept of an "event", which emerges as the logical unit of analysis and becomes the basis for describing intelligence information; and presents the "template" as an event-centered information structure.

Section 3 describes the design and implementation of the Event Representation Language (ERL), which is centered around the notion of "template" and embodies both the declarative and procedural knowledge requisite for the complete representation of events and their properties. ERL is embedded in the programming language Prolog, an interactive programming language based upon a simple proof procedure involving a subset of classical logic. The method of encoding templates in Prolog is discussed in detail. This is followed by an explanation of the ERL procedures developed for Event Record Synthesis, and a detailed example of how ERL template representations accomplish the semantic interpretation process, that maps the syntactic structures output by the parser onto event records.

Section 4 comprises an overview of MATRES II, which is implemented on the PDP 11/45 under RSX-11D. The base language for all the programs -- except the ERL compiler -- is Forth. The ERL Compiler is coded in SPITBOL, a dialect of SNOBAL 4, which was chosen because of its excellent facilities for compiler writing.

Part II of the report presents a detailed description of the implementation of MATRES II. Its data structures and algorithms for the sentence input and grammar processing vocabulary and the capabilities in the area of morphology are described. The implementation of the ERL evaluation process, including the abstract machine which is the target language of ERL, is described in Section 4. The ERL Compiler, which is the only non-Forth module, is discussed in Section 5.

A supplementary report not for publication is on file.

TABLE OF CONTENTS

EVALUATION

The main objective of the work described in this report was to develop a computer-based technology which would substantially assist the I&W analyst in reviewing and processing the contents of large volumes of message traffic.

In particular, this effort focuses on providing a detailed conceptual and methodological framework for an advanced event processing system designed with the aim of distilling significant information elements from the narrative text of intelligence messages and synthesizing fixed-format, problem-oriented information structures in support of I&W data base generation and update functions. These information structures present information to the analyst in a compact and usable format thus reducing his burden and making it easier for him to concentrate on higher-level analytical activities. Specifically, the work described addresses the issues involved in synthesizing meaning representations from a particular class of intelligence messages constituting reports related to the air activities domain, used by the advanced Indicator System (AIS) at the NMIC.

The task of teaching a computer to "understand" language and identify relevant information elements is not a trivial one. It requires the utilization of advanced technologies involving formal syntactic and semantic analyses of the sentences of a text, and the development of techniques for synthesizing appropriate meaning representations in the form of machine-processable information records.

A system -- designated as a Message Analyzer Testbed and Results Evaluation System (MATRES) -- is presented in the form of three major components: (1) The Lexical Unit Recognizer, (2) The Augmented Transition Network Parser, and (3) The Event Representation Language Machine. Taken together, these components analyze incoming textual reports of events and, from them, synthesize "event records" (i.e., extract relevant information and store it in event-centered information structures utilizable as data base records). The technique employs frame-like "event templates" for representing general knowledge about event classes. These are essentially intensional descriptions of events and, as embedded in the system, they behave as active data structures which drive the synthesis process. The system, as currently conceived, provides an adequate conceptual basis for a generalized text "understanding" system capable of dealing with intelligence reports describing events involving movements and activities of objects comparable to aircraft (i.e., missiles, satellites, ships, submarines, etc.).

One of the notable features of the system is the use of the programming language Prolog, a formalism based upon a subset of classical logic, which lends itself particularly well to the encoding of the logical argument structure of event descriptions. Recent investigations reported in the literature show that Prolog is not only used for the grammatical description of structures and processes of natural language, but can also be used as a practical tool and a unifying principal for the description and manipulation of data bases. The use of Prolog, therefore, deserves attention in any further investigation related to automated data base generation.

In conclusion, the approach taken in this investigation is a significant step in providing a framework for a system whose main purpose is to map narrative text into information on structures in support of I&W functions.

[illegible]

ANDREW S. KOZAK
Project Engineer

1.0 Introduction and Summary

1.1 Introduction

This report describes an RADC sponsored contractual effort related to the development of automated analytical tools in support of the I&W analyst.

The task of an intelligence analyst is to predict the future on the basis of information describing what has happened in the past and what events are currently taking place. The basic information source for most analysts is intelligence messages, which come in large volumes from many different originators, largely in the form of narrative text.

The questions the analyst asks himself are: "What is happening?" "What does it mean in terms of my knowledge about similar events in the past?", "What is going to happen next?" He is concerned with certain states of affairs, and events signifying changes in these states of affairs. His evaluations of incoming information are based on his cognitive models of such states of affairs, the personalities, entities, and processes involved, and the potentialities and constraints associated with changes in an existing state of affairs.

Given the volume of information he must sift, and the complexity of the cognitive models involved, the difficulties of the analyst's task are obvious. Aids to support his analytical processes clearly must involve means for distilling the content of incoming information into a form which is compact, usable, and compatible with his view of the world.

The work described here is concerned mainly with the development of a technology for the automated analysis of unformatted (free-text) with the aim of transforming it into fixed-format, problem-oriented records, reflecting its information content. The subject domain under investigaton is that of air activities.

When reviewing narrative text, the human analyst uses his innate knowledge of English grammar, as well as his extra-linguistic knowledge of entities such as aircraft, time, location, and actions -- including all the relevant concepts which can be attributed to or are implied by such entities -- and extracts those information items which are relevant to his task.

In order to model this human cognitive activity, the computer must be equipped with representations of both linguistic and extra-linguistic knowledge, and a means of manipulating such representations for the analysis of text and synthesis of information elements. The elements must then be presented in a clear and useful format suitable for the task at hand.

The approach adopted by OSI is based upon current advances in language "understanding" by computer, as exemplified by work in Computational Linguistics, Artificial Intelligence, Cognitive Psychology, and non-numeric programming technology. A survey of the field as related to the work reported here can be found in Silva and Montgomery (1978).

Briefly, OSI's approach to the problem combines a "bottom-up" data-driven analysis based upon linguistic and logical principles with a "top-down" conceptually driven domain-specific interpretation of the structures generated by the input analysis. The "bottom-up" analysis is carried out by an augmented transition network (ATN) parser, which uses a dictionary and a grammar of the reporting language to produce a parse tree showing the constituent structures of the input string and their hierarchical relationships. The interpretive procedures are embedded in the Event Representation Language (ERL), which uses domain-specific knowledge stored in permanent data structures called "templates" to transform the linguistic structures generated by the parser into template-derived content representations.

1.2 Summary

This final report presents the results of the exploratory and developmental work performed under this contract. Briefly, the work involved extensions and additions to the simple ATN grammar constructed under a previous contract to accept a wider range of linguistic structures; the refinement of the notion of "template" as a data structure for the representation of knowledge about events; the design and implementation of the Event Representation Language, a language written to explore the use of the "template" as a knowledge representation technique with which to build systems for automated language analysis; and finally, the construction and implementation of algorithms for the automated analysis of narrative text and its subsequent transformation into formal content representations.

A major effort was devoted to the implementation of MATRES II, the OSI message text analysis system, which incorporates the technologies mentioned above, and involves the ability to digest narrative text and systematically transform it into concise, machine processable content representations, called 'event records', in which a message can be viewed from several perspectives: time, location, organization involved, activity type, etc.

Table 1-1 shows an input sentence and the corresponding event record produced by MATRES II.

The report is divided into two major parts. Part I deals with system design, while Part II describes the implementation of MATRES II.

Section 2 of Part I describes some of the fundamental concepts underlying message text analysis. Section 2.1 describes the structure of event reports viewed from different perspectives. Four levels of description are distinguished, each corresponding to a major processing phase.

In Section 2.2, the 'event' emerges as the logical unit of analysis and becomes the basis for describing intelligence information. Events have a complex internal structure and raise special representational issues. OSI has given particular consideration to this question and has developed an event-centered information structure called a "template", which lends itself particularly well to the description of events and their associated concepts. In Section 2.3, the template is first described from the point of view of the user, stressing those properties which render it particularly useful as an analytical aid. This is followed by a description of the internal structure of the template.

Section 3 describes the design and implementation of the Event Representation Language, which is centered around the notion of "template" and embodies both the declarative and procedural knowledge requisite for the complete representation of events and their properties. It has the additional advantage of being readily processable by computer.

ERL is embedded in the programming language Prolog, an interactive programming language based upon a simple proof procedure involving a subset of classical logic. We were fortunate to discover this language at a time when we were searching for a good representation for the template concepts that we were developing in an intuitive and informal way. Prolog gave us not only a natural and perspicuous notation for the uniform representation of the template concepts, but also provided a feasible and fairly efficient

Table 1-1 Example Input and Output by MATRES II

```
+-----------------------------------------------------------------+
|                                                                 |
|                             Input                              |
|                                                                 |
|                                                                 |
| TWO UGANDAN AIRCRAFT FROM REGIMENT A1313 AT ENTEBBE             |
| DEPLOYED TO GULU AT 0200Z ON 21 FEBRUARY.                       |
+-----------------------------------------------------------------+


+-----------------------------------------------------------------------+
|                                                                       |
|                              Output                                   |
|                                                                       |
|                                                                       |
|     Event: DEPLOY                                                     |
|     Object:                                                           |
|     ...Equipment= UGANDAN ACFT                                        |
|     ...Nationality= UGANDAN                                           |
|     ...Subordination= FROM REGIMENT A313                              |
|     ...Stagingbase= AT ENTEBBE                                        |
|     ...Number= TWO                                                    |
|     Destination= TO GULU                                              |
|     Time= AT 0200Z                                                    |
|     Date= ON 21 FEBRUARY                                              |
|     EVENT RECORD COMPLETE.                                            |
|                                                                       |
+-----------------------------------------------------------------------+
```

implementation approach.

Section 3.2 illustrates the method of encoding templates in Prolog. This is followed by an explanation of the ERL procedures developed for Event Record Synthesis (3.3), and a detailed example of how ERL template representations accomplish the semantic interpretation process, which maps the syntactic structures output by the parser onto event records (3.4).

Section 4 comprises a brief overview of MATRES II, a description of the scope of the linguistic grammar and lexicon developed for the aircraft domain, a description of the MATRES II parser, and a discussion of some fundamental issues underlying the selection of template descriptors for a particular subject domain. It concludes with the list of the descriptors so far identified for the air activities domain.

MATRES II is implemented on the PDP 11/45 under RSX-11D. The base language for all the programs -- except the ERL compiler -- is Forth. The ERL compiler is coded in SPITBOL, a dialect of SNOBOL 4, which was chosen because of its excellent facilities for compiler writing.

Part II of the report presents a description of the implementation of MATRES II. Section 2 of Part II describes the data structures and algorithms for the sentence input and grammar processing vocabulary; this vocabulary is essentially an extensive modification of the MATRES I system. Section 3 describes the capabilities added in the area of morphology. The implementation of the ERL evaluation process, including the abstract machine which is the target language of ERL, is described in Section 4. The ERL compiler, which is the only non-Forth module, is discussed in Section 5. The last three subsections are intended as a guide to the Forth program files listed in Appendix A, and provide glossaries of the Words in those files.

Appendices A-G contain program listings of all the MATRES II modules (A), an introduction to the programming language Prolog (B), a listing of the aircraft domain lexicon (C), a listing of the current version of the ATN grammar (D), a sample listing of sentences now parsed by the MATRES II System (E), a set of examples of system input/output (F), and operating instructions for MATRES II at OSI(G).

1.3 Conclusions

The method of approach which OSI has adopted since the inception of the RADC Exploratory and Developmental program for Automated Data Base Generation has been to look ahead to the potential capabilities of a future system for both interactive and fully automated exploitation of the narrative text of intelligence messages, and to develop a methodology that will remain valid for applications of considerably greater scope than the one currently under development.

Although MATRES II is at an early stage of development, it has demonstrated that OSI's initial design concept was sound, and can eventually be developed into a useful operational tool in support of I&W functions. The concepts underlying its design and implementations appear so useful, that the system has already aroused considerable interest both within and outside the intelligence community.

One of the noteworthy features of MATRES II is its modular design, which has greatly facilitated its implementation. In spite of its complexity, MATRES II was written by a single programmer working only half-time in about one year.

There are two aspects of MATRES II as a language "understanding" system that make it somewhat unusual, and thus deserve particular mention: first, it operates on a 16-bit minicomputer, within a quite limited amount of available memory (64K bytes); second, it is written not in one of the popular AI extensions to LISP, but in Forth, a language designed for use on small minicomputers which combines a low-level, machine-oriented semantics with a natural facility for extension of the semantics in a user-defined way. It was basically those combined properties of Forth, together with its fairly simple "virtual memory" facility, that has made it possible to implement such a structurally complex application on a small machine in a relatively short time. As it currently exists, the MATRES II system fills the available memory with only a small amount of dynamic space available for sentence processing; this can be ameliorated with a modest amount of work, at the cost of noticeably slowing the processing due to virtual memory I/O.

2.0 Message Text Analysis: Fundamental Concepts

In this section we explicate some fundamental concepts related to message text analysis. We begin by a formal description of event reports, which constitute the primary data of our analysis programs. Next, we consider the notion of 'event', which emerges as the primary unit of analysis, and becomes the basis for the design of the Event Representation Language described in detail in Section 3. Finally, we present the concept of a 'template', first as an analytical aid designed to support the analyst in his task, and second, as an information structure which provides an event-centered framework for the uniform and compact description of event data contained in intelligence messages.

2.1 The Structure of Event Reports

Work under previous contracts has shown that the formal description of event reports requires a multi-level approach. Four levels have been identified to date, each involving a different aspect of event reporting, and each based upon different considerations.

2.1.1 The Macro Level. This level of description involves the composition of reports in terms of individual messages and is based upon operational considerations.

An 'Event Report' is defined as a collection of one or more messages transmitted over a period of time and dealing with the same event. For example, an event report concerning a specific flight might consist of three messages M1, M2, and M3. M1 might describe the flight of some as yet unidentified aircraft over some general area. A second message M2 might request a change of any one of the flight parameters reported in M1, while a third message M3 might be a follow-up, adding new information to the aircraft first reported as 'unidentified'.

If all parameters of an event are clear to the observer at the time of reporting, and can therefore be reported with certainty, the description of the event usually involves only one message.

From the point of view of automated computer analysis, a distinction must be made between those messages that contain new event descriptions (i.e., descriptions of events reported for the first time), and those that either request changes in the parameters of some previously reported event, or add information to previously underspecified parameters. From an operational point of view, a first report involves creating a new data element, while requests for change and updates involve changes and/or additions to an already existing structure.

Let us call the class of messages that lead to the creation of new data elements class M0, and those that imply changes to the data base class M1. The composition of event reports at this level can now be formalized in the form of a grammar using BNF notation:

```
(1)    <Event Report> →  <Message>| <Messagelist>
       <Messagelist>  →  <Message> | <Messagelist><and><Message>
       <Message>      →  M0, M1
```

Work under this contract has focussed mainly on messages of class M0, i.e., reports of new events. In the next section we examine the structure of such messages from the point of view of their information content.

2.1.2 The Message Level. This level of description involves the composition of single messages in terms of classes of events characteristic of a particular subject domain. In the following paragraphs we shall be concerned only with messages of class M0.

Messages can have a complex internal structure comprising header information, followed by either formatted, semi-formatted, and/or unformatted (narrative) text portions, before ending with some special symbols signalling the conclusion of the message.

Since this work is concerned mainly with the narrative text portions of messages, we shall describe the messages in terms of three components: a 'pre-text' component, the 'text' component, and a 'post-text' component.

The 'text' component of a single class MO message may contain the description of one or more new events. Of course, not everything reported in a message text is a description of an event. There are also objects, and states, and processes, and perhaps other entities. However, events are of fundamental importance and it is expedient to treat object, state, and process descriptions as special types of events. (For a full discussion of the concept of an 'event' see Section 2.2).

Thus, a message may state that a particular set of aircraft carried out a penetration flight over some country, and then engaged in some other activity before returning to homebase. Such a message describes a chain of connected events: a penetration flight, followed by an activity, followed by a return to homebase, all involving the same set of aircraft. The events reported on in a message need not always be connected as described in the previous example. It is quite possible for a message to report on several seemingly unconnected events.

The content of a message can now be formally characterized in terms of the 'event' as the primitive unit:

```
(2) <Message>   → <Pre-text> <Text> <Post-text>
    <Text>      → <Event> | <Eventlist>
    <Eventlist> → <Event> | <Eventlist> <and> <Event>
    <Event>     → e1,e2,e3,.....en
```

Note that <and> is a symbol of the metalanguage and represents a set of operations and relations on events, while the set of event classes characteristic of the subject domain covered by the class of reports 'Event Report' defined in (1) above is symbolized by {e1,.....en}.

We shall call (2) the 'Message Grammar'. It consists of a designated non-terminal <Message>, called the 'initial' symbol, a set of non-terminal symbols Vn, the set of given symbols Vt, called terminals, and the four productions in (2).

Vn = Message, Pre-text, Text, Post-text, Eventlist, Event, and
Vt = {e1,e2,e3,.......en}

The set {e1,e2,.....en} comprises the 'primitives' of the 'Message Language' described in (2), and will vary from subject domain to subject domain.

2.1.3 The Event Level. This level involves the description of events in terms of their properties, including time, location, action, and object(s) involved in the action.

Event descriptions take two forms: *intensional* descriptions, and *extensional* descriptions.

An intensional description is an abstract description of a class of *individuals* in terms of a set of invariant properties common to all members of the class. Thus the intensional description of the class of flight events would state that all such events are associated with objects that can fly, have a specific location at some point in time, may involve a mission, and can further be specified in terms of path, altitude, direction, and extent of flight. It would also state any associated inferences, such as, for example, that a flight event is necessarily preceded by a take-off event.

An *extensional* description involves one individual, i.e., a unique member of a class of individuals in the world being modeled. *A simple example is the description of a specific* aircraft (e.g., a MiG-21) flying in a given direction (e.g., north), at a particular time (e.g., 0100Z).

The representational construct developed for the description of events is the 'template'. It is an abstract data structure containing a collection of invariant information reflecting the analyst's view of the concept it describes. All information represented in templates is associated with rules governing its use.

The class of individuals to which an intensional description applies is called the *extension* of the general concept described by the template. In the context of Event Language Recognition, descriptions of individual events are called 'Event Records'. Thus, the set of event records describing events of the same class, i.e., event records related to a particular template, constitute the *extension* of the concept described by the template.

Section 2.3 gives a general overview of the template from the the viewpoint of the user and stresses those features which can serve as an aid to the analyst. This is followed by a detailed study of the internal structure of the template as the fundamental knowledge structure for the representation of event data. The criteria for the selection of the descriptors appropriate for a particular subject domain, and the methodology employed as applied to the aircraft domain are discussed in detail in Section 4.5.

2.1.4 The Linguistic Level. This brings us to the fourth and last level of description discussed here, namely to the description of the linguistic structure of narrative text. In the MATRES II System, the linguistic structure is defined by means of an augmented transition network grammar in terms of familiar linguistic categories such as sentence, nounphrase, verbgroup, prepositional phrase, adverb, and others.

In order to expedite processing, a number of language specific categories, not usually found in traditional grammars, were added. Thus, the familiar definition of prepositional phrase in (a) was augmented to encompass dates (b) and times (c):

```
(a) pp → prep + nounphrase
(b) pp → prep + date
(c) pp → prep + time
```

where 'date' and 'time' are non-terminals of the grammar with their own internal structure. The MATRES II grammar and its associated lexicon are described in detail in section 4.3 of this report.

2.2 The Concept of an Event*

Although the event concept is fundamental in message analysis, no standardized terminology for describing or classifying events exists. Thus, when reference is made to the parameters of 'event/time/location,' the event concept used is that of a type of

* This concept was originally developed in Kuhns, et al. (1975) under a previous RADC contract.

activity. In another usage 'event' refers to a fact. The event concept in physics is that of a point in the spacetime continuum, and in mathematical statistics the word 'event' has the broadest meaning, that of any proposition, whether true or not. The event concept has even been taken as a primitive (i.e., as undefined) and then used to define the series of time points (Weiner 1914; Russell 1956).

To deal with event reports it was necessary to define a 'data semantics' and a corresponding *Event Representation Language* which, as the name implies, is a special language for the description of events and event-related concepts, such as objects, processes, and other entities. This language also has the desirable feature of being representable in an appropriate formalism (see Section 3), which is amenable to computer processing. This language guides the mapping process which converts narrative text into formatted event records (see Section 3.4).

By an event we mean roughly either the property that an object has at a point in time or over a time interval, or a relation that holds among a set of objects or locations at a point in time or over a time interval.

Events may be gathered into certain event classes called activities, e.g., air activities, submarine activities, and ground activities. These are characterized as involving certain types of objects or properties or relations.

We give a classification of events based on considerations involving sources of reports, observers of events, and relations and properties involved in events. Time points are the central element of an event. A discussion of time data as it occurs in natural language is given in Kuhns and Montgomery (1973) and Kuhns (1975).

The most complicated examples arise in messages containing narrative text. These illustrate the variety of problems that arise in defining events and the necessity for their classification. One example is:

> A reliable source reported that water tankers
> accompanied by trucks carrying what appears to
> be ... have been observed stopping .... This
> could indicate that ... .

The initial analysis of the first sentence shows that this involves four levels of events. There is first the fact that water tankers were accompanied by trucks, etc.; there is second, the *observation of the fact;* third, there is the *report of the observation* (i.e., via the referenced source); and finally, there is the message itself which is a *report of a report*. To distinguish these levels, we introduce the notion of a meta-event.

We define a meta-event to be a report of an event. There are two sorts of events other than meta-events: an *observational event* which is a direct perception of an event (often indicated as a visual perception, e.g., 'observe,' 'sight'; an electronic perception, e.g., 'contact'; or a term ambiguous as to the nature of the contact, e.g., 'identify,' 'detect');* and a *primitive* event which does not involve an observation. The meta-events themselves are distinguished by *orders*. A meta-event that reports on an observational or primitive event is a *first order meta-event*.

A meta-event that reports on nth order meta-events is an (n+1)th order meta-event. Thus the previous example message is a second order meta-event reporting a first order

---

* A special case of an *observational event* is a *designation event* where a special proper name for an object is introduced, e.g., 'an aircraft, designated as MiG-21,...'

meta-event consisting of an observational event of a primitive event.

A similar analysis can be given for the example:

> It is reported that two Ugandan F class fighter aircraft
> were sighted in the vicinity of 0200S3230E at
> 011200 hours.

Here, the originator of the message uses a passive sentence construction rather than stating the source of the report. This too is a second order meta-event describing a first order meta-event describing an observational event of a primitive event.

The meta-events can be considered to convey certain pragmatic information that is of importance to the intelligence analyst. The most important aspect of this pragmatic information relates to the credibility of the event data being reported. Other aspects are fact of the message transmittal itself, the time and origin of transmittal, etc. The decision making function related to the pragmatic information involved is one of feed-back, just as for other sensors: to reorient the data collection, to suspend it, or amplify it.

Since a message is itself a report of an event, it is a first order meta-event.

A further breakdown is used for primitive events. We distinguish two kinds. An *attributive event* gives a situation where a particular object* or object of a certain type has a certain attribute (other than spatial location), which may be inherent or temporally constrained: i.e., the attribute is true at a certain time, or in a certain time interval.

Thus, the example:

the aircraft is Y-class

is an attributive event. In the notation of the predicate calculus an attributive event is symbolized as:

$$P(x,t) \qquad (1)$$

where x is the object argument, t is the time argument and P is the attribute that x has at time t.

The argument-expression 'x' can have a variety of forms through which additional properties of the object can be expressed -- chiefly through the use of descriptive phrases.

The second kind of primitive event is called a *relational event*. This is a situation where n objects or an object and a location stand in some relation to each other at a certain time. Position data may be absent, but when it occurs with one object-argument, then the relational event gives either the location relation holding between an object and its position at a certain time (e.g., the primitive event described in the second example) or some other relation between n objects(e.g., 'the aircraft entered Biafran airspace').

A relational event is symbolized as:

$$R(x1,....,xn,t)$$

---

* Objects are taken in the broadest sense and include cultural objects (such as governments, institutions, etc.), and psychological objects (perceptions, attitudes, etc.).

where x1,....xn are object or location arguments, and t is the time argument. Among the relational events is a class of special interest -- these are events giving a *world point* of an object, i.e., its space-time coordinates. Such an event is called a *location event*, or *world point event*.

Thus, the track of an aircraft, which consists of a set of world points, is a portion of its world line. For a location event, the expression (2) then takes the form:

$$L(x,p,t) \qquad (3)$$

where p is the location argument.

Another class of relational events is given by a generalization of (3). These we call *world point qualifications* or *location-event-qualifications*. Such an event is a two-place relation (other than location) between an object and a location that holds at a certain time. Thus, the relation stipulates the activity of an object at a certain point and time, e.g., an aircraft flying south in the vicinity of ... at ... . In symbols this is expressed as:

$$Q(x,p,t) \qquad (4)$$

where Q is the activity engaged in by x at the space-time point (p,t).

In an event record we can consider the world point of an object to be a property of the object. This property can be defined formally through use of the λ-operator which is used in logic to introduce new predicates (i.e., names of attributes or relations). Thus, the world point property corresponding to (3) is written as:

$$P=(\lambda x)\ L(x,p,t) \qquad (5)$$

or as coordinates:

$$(p,t) = (\lambda x)\ L(x,p,t) \qquad (6)$$

For example, in the second example above:

(p,t) = (in the vicinity of 0200S 3230E, 011200)

Similarly, a world point qualification can be expressed as a triple giving the space-time point of the object and its attribute. For this we write:

$$(p,t,Q) = (\lambda x)\ Q(x,p,t) \qquad (7)$$

An example of a world point qualification would be:

(p,t,Q) = (0200S 3230E,011200, moving not at all)

In all these formulations, we are treating the time arguments on a logically different level from the object and location arguments. The reason for this is that time arguments always occur in event formulations (even though sometimes only implicitly) while other arguments need not occur.

If an event involves two or more objects or locations it is called simply a non-world point event, because it is not uniquely classified by object and location. However, many such events can be reduced into world point events or qualifications. For example, 'John met Mary at ...' can be reduced by use of the λ-operator) to either a property of John (and hence a world point qualification event) or a property of Mary (similarly). Also, for example, 'The aircraft flew from London to Paris' splits into two world point events for the same object -- a departure at London and a (later) arrival at Paris.

A summary of the classification of events is given in Table 2-1.

In the symbolism above, we have broken a primitive event into object, location, and time-arguments and relations and properties. Indeed, every situation can be so analyzed. This has been referred to in the literature as thing-splitting [Reichenbach, 1947]. It is often more natural to introduce events themselves as arguments -- for example, in the analysis of meta-events or observational events. Thus, a meta-event can be symbolized as:

$$R(s,e,t) \tag{8}$$

which describes a report of e by source s at time t.

Similarly an observational event is:

$$O(x,e,t) \tag{9}$$

which describes the observation of e by the observer x at time t.

Table 2-1. Classification of Events

- Meta-events
- Non-meta events
  - Observational events
  - Primitive events
    - Attributive events
    - Relational events
      - World point events (location events)
      - World point qualification events (location event qualifications)
      - Non-world point events, or events involving two or more objects or locations

The introduction of events (even those corresponding to primitive events) as arguments in situations can be achieved by certain symbolic devices. This is called event-splitting (Reichenbach,1947), i.e., the situation is 'split' conceptually into an event-argument and an event property. Language has various devices for event-splitting, the chief one being nominalization, e.g., 'arrival,' and the use of the word 'that' which flags an event-argument.*

Primitive events can be further concatenated into *event chains* which are a sequence of minor events giving an initiation, continuation, or termination of certain major events. For example, the major events of aircraft penetration could include such minor events as the initial penetration of airspace, the reaction, and a termination such as the departure or destruction of the aircraft. Observational events can also be gathered into event chains. Establishing, maintaining, losing, and regaining contact with an aircraft provides an example of this. Another use of event chains is the monitoring of minor events to forecast major events. The definition of *interval events*, i.e., activities which continue over an interval of time (as opposed to *point events*), can be accomplished by reducing them to point events. This is accomplished by defining an event type, such as flying,

* A formal method for introducing event arguments is described in Kuhns (1975).

and then stating that an event of this type occurred at every point in a time interval. In English, interval events are usually associated with the progressive tenses. (Verb tenses furnish important implicit time data of a relative nature (Kuhns and Montgomery (1973) and Kuhns (1974).

While an observational event involves the perceptional facility of an observer, there are other events involving the evaluative facility of an observer, source, or person. These are events dealing with appraisals of an object property, a truth value for occurrence of an event, and a degree of belief in an event. An example is given by the first message sentence (above). Thus, the phrase 'what appears to be' flags the evaluation of an object-property.

The phrase beginning the second sentence 'This could indicate that' flags a hypothetical event, i.e., one that is not asserted as occurring but only as possible or predicated. Similarly, the phrase 'a reliable source' is an evaluative component of the message. It would seem that these evaluations should be distinguished from affirmative or direct assertions such as 'determined to be,' 'confirmed to be,' 'the previous report is erroneous' even though these can be considered as extreme cases of evaluations, just as any report can be so considered. We can say that an evaluative component of an event, or an evaluative event itself (e.g., this could indicate that ...), is one that expresses the reporters subjective judgment of the information conveyed. These notions are tentative and should be subject to further study. For the time being, we class an evaluative event as a special kind of meta-event. It seems that an evaluative component of an event, such as the appraisal of an object-property, can be analyzed as the conjunction of an evaluative event and a primitive event, and that this approach can be consistently carried through.

2.3 The Concept of a Template

In this section the concept of a template is explicated from two points of view. First, the focus is on those properties which render it particularly useful as an analytical tool; then, the focus shifts to its internal organization as a knowledge structure for the representation of event data.

2.3.1 The Template as an Analytical Tool. In this section we present a general view of the template as an information structure for the description of event data with particular emphasis on those features that render it useful to the analyst in his task.

The template as the basic knowledge structure for the compact and uniform representation of information on entities and events described in intelligence messages provides the means of coding the analyst's cognitive models in terms of logical data structures which are susceptible to automated processing.

An event template is composed of a set of information parameters or descriptors which represent the type of information that answers the set of questions shown in Table 2-2, which also illustrates the corresponding descriptors of a prototype template.

This is somewhat of an oversimplification of the template concept for convenience of presentation, since complex descriptors within the template are actually represented by pointers to other types of templates: e.g., object templates. Thus for the message given in example a), an object template reflecting an aircraft description is essentially embedded in the event description by a pointer reference, as shown in Table 2-3:

TABLE 2-2. Information Parameters of a Prototype Template

QUESTION	DESCRIPTOR	EXAMPLE
what	event type	airspace
who	agent (or object plus owner)	a Ugandan MIG-21
when	time of event	at 0235Z 25 April 1978
where	location of event occurrence	6 miles from the Kenya border near Suam
to whom	patient, or entity affected by the event	probable reconnaissance mission

(a) TWO SAAF CAPETOWN-BASED SAG22 ACFT ARE OPERATING OVER THE INDIAN OCEAN.

It is interesting to note that message a) is incomplete in terms of the prototype template specification outlined in Table 2-2. Conspicuously absent is a time descriptor.*

For the formal description of the event in (a) to be complete, a time descriptor element must be satisfied. The absence of this element can be signalled to the analyst to indicate that this element must be supplied for the information representation to be complete. Thus, *intra* -template relations -- the set of relations connecting descriptors *within* a template -- provide an important means of alerting analysts to the missing information elements in data structures constituting subparts of the network of templates which represents an analyst's cognitive model of a state-of-affairs.

Another set of relations which can be very useful to the analyst are relations between templates, or *inter* -template relations.

For example, an aircraft cannot fly unless it has taken off, cannot land unless it has been flying, must be in flight if it has taken off and has not landed or been destroyed. Thus a TAKE-OFF is a template which represents a necessary predecessor event to a FLIGHT event. On the other hand, a LAND event is a possible, but not obligatory successor to a FLIGHT event, for the object involved in the flight may have changed course, or may have been destroyed before landing.

Inter-template relations predict the normal, expected, ordering of events in the air activities world. Any violation of these expectations can serve as a warning to the analyst that some external force has altered the predicted course of events. It is therefore important that the analyst be alerted to any deviation from the expected.

---

* Also a separate template because of the complexity of most time descriptors, which are derived only partially from explicitly stated times as in the Table 2-2 example; they must often be reconstructed from the tense of the internal verbs, time operators such as 'currently', which point to information in the message header or the textual context of the time referent, and the internal structure of a time descriptor, which may read 'at 10 minute intervals for a period of 6 hours'.

Table 2-3 Content Representation for Sentence (a).

```
+-----------------------------------------------------------------------+
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
|                       EVENT DESCRIPTION                               |
|                                                                       |
|  UNIT REPRESENTED:     EVENT                                          |
|  EVENT TYPE:           BE ACTIVE                                      |
|                                                                       |
|  OBJECT:-------------------------------------+                        |
|                                              |                        |
|  REGION:               THE INDIAN OCEAN      |                        |
|                                              |                        |
|                                              |                        |
|                                              |                        |
|                                              V                        |
|  +----------------------------------------------------------------+   |
|  |                                                                |   |
|  |                                                                |   |
|  |                    AIRCRAFT DESCRIPTION                        |   |
|  |                                                                |   |
|  |  UNIT REPRESENTED:       OBJECT                                |   |
|  |  OBJECT TYPE:            AIRCRAFT                              |   |
|  |                                                                |   |
|  |  TYPE:                   SA622                                 |   |
|  |  SUBORDINATION:          SOUTH AFRICAN FLEET AIR FORCE         |   |
|  |  BASE:                   CAPETOWN-BASED                        |   |
|  |  SET SPECIFICATION:      TWO                                   |   |
|  |                                                                |   |
|  +----------------------------------------------------------------+   |
|                                                                       |
|                                                                       |
+-----------------------------------------------------------------------+
```

Accordingly, the explicated network of *inter* template relations, both obligatory and optional, provides an additional means of alerting the analyst to the implications of an event, as well as to related events which may furnish data elements missing in the template currently being processed.

In summary, the application of template technology to the analysis of narrative text can provide an important analytical aid from the following five points of view:

- Templates constitute a powerful means for distilling the content of verbose textual messages into a compact format.
- Templates are logical information structures which can be used to represent the analyst's cognitive models of events and states of affairs.
- Inter and Intra template relations can assist the analyst in recognizing missing elements of information and predicting future events.

- Templates provide a discrete representation of an event which lends itself readily to statistical analysis, as in indications monitoring applications for I&W.
- Event data available from the TIME and LOCATION descriptors of templates can be exploited to drive automatic plotting of ship, submarine, and aircraft tracks.

2.3.2 *The Internal Structure of the Template.*

Information contained in templates is of two types: declarative information and procedural information. We begin our discussion with a preliminary specification of the declarative elements of the template and the relations between those elements. Next, we present the procedural components of the template and consider the operations they perform.

2.3.2.1 Structure of the Template: Descriptive Elements Essentially, a template is a data structure which has a unique "name" and an ordered set of "slots" filled by "descriptors". Template "names" serve as identifiers and refer to the entity described by the template. Examples of template names in the air activities domain are: BE ACTIVE, FLY, DEPLOY, ARRIVE, AIRCRAFT, and DTG (date-time group).

The substructure of a template and its relations to other templates is defined in its descriptor slots. Descriptors bear special meaning relations to the central concept. Each descriptor slot names some property of that concept. For example, most event templates involve the descriptor slot "Object", which names the relation of the entity which fills this slot to the event. In general, descriptors are of several types. Some may assign an object to membership in a category (such as "is an aircraft"); others may state an object's functional role in a complex event (the "Source" of a particular flight); yet others may express the time and place of an event ("at 0115Z"; "along the River Kwai").

Each descriptor slot has a name which is unique within the template. Associated with each descriptor slot is a set of one or more statements which constrain what may "fill" the corresponding slot in the representation of an individual entity. These statements will be referred to as "filler specifications". Filler specifications then, give linguistic information, i.e., they specify how a particular descriptor can be realized in the sublanguage. A filler specification indicates the possible deep-structure syntactic environments for a given descriptor as well as the properties of the items to which the rules which map syntactic trees onto meaning representations are sensitive. Thus, the description associated with the 'Object' descriptor in the DEPLOY template for the air activities domain, would specify that the object is normally an aircraft. In the description of an individual event of the DEPLOY class, this is further specified as some specific aircraft, (e.g., a Nairobi-based F-4 Phantom fighter). The descriptors associated with the DTG template, on the other hand, specify the permitted range of values for its components (see Table 2-4). For example, the day in a particular month must lie within the lower limit of 1 and an upper limit equal to the length of the corresponding month. A day number of 77 would be outside the permissible range of variation for this descriptor. If such an anomaly is discovered, it must be brought to the attention of the analyst operating the system.

Any descriptor of a template can be defined as a separate template when its internal description is important to the analyst and is of sufficient complexity to warrant a separate representation. For instance, the template for the DEPLOY event class incorporates a descriptor "Object", whose attribute, in this template, is "aircraft". But "aircraft" in itself is a complex notion of I&W significance and is represented by a template of its own (see Table 2-3 above).

The template as a unit for representing knowledge, therefore, is complex and extensive. Rather than being of the order of a single property or relation attributed to the entity described, it is an n-place hierarchical relational definition of a concept with optional pointers indicating relations with other templates.

In summary, the descriptive elements incorporated in a template represent the analyst's knowledge of concepts and their interrelationships in his particular task domain.

Table 2-4. Description of the Date-Time Concept (DTG)

TEMPLATE NAME	DTG (DATE-TIME GROUP)
DESCRIPTOR NAME	FILLER DESCRIPTION
Day-number	First two digits of seven character string of format NNNNNNZ. Constraints: max day number = Month Length
Zulutime	time concatenated with "Z".
Time	Four digit string with constraints: 0000 < Time < 2400
Month	Three character string. Member of set (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sept, Oct. Nov, Dec).
Year	Two digit string constraints: 1st digit = 7 2nd digit = 9

2.3.2.2 Structure of the Template: Procedural Components As mentioned previously, templates are active data structures which incorporate both declarative and procedural knowledge. This section is concerned with the procedural components of templates and how these are used by the system throughout the process of narrative text analysis.

Procedures are attached to descriptor slots. They are essentially mapping rules which effect the transformation of parsed sentences into event records. Procedures carry out specific computations. Some define the steps involved in finding "fillers" for particular descriptors, others specify the operations involved in identifying the referents of anaphoric expressions, while yet others may compute relations between events.

These mapping rules are necessarily language specific. They incorporate the domain-specific pragmatic knowledge which establishes the link between the abstract description of an event class (the template) and the description of an individual member of that class (the event record).

The process of "understanding" a sentence consists of an interaction between the procedures associated with the descriptor slots of the corresponding template, each of which actively seeks to satisfy its own requirements. Essentially this is done by searching the parse tree for constituents which satisfy the syntactic and semantic constraints on the permissible fillers for a particular descriptor.

It should be noted here that not all descriptor slots in a template need be filled for any particular input sentence. A template provides slots as placeholders for information that is considered relevant, even though it may not always be present in the input. The number of slots of any template that will be filled, therefore, depends largely on the information contained in the input sentence. It is important to note, however, that, for a sentence to be considered as "understood", the following two conditions must be met:

- Every element of the input sentence must be assigned to some descriptor slot;
- All "obligatory" descriptor slots must be filled.

3.0 The Event Representation Language

3.1 Introduction

The Event Representation Language (ERL) developed under this contract is an experimental language especially written to explore the use of "templates" as a knowledge representation technique with which to build systems for message text analysis in support of I&W functions. The basic data objects of ERL are the templates.

ERL is implemented in a subset of Prolog, an interactive programming language based upon a simple but efficient proof procedure involving a subset of classical logic referred to as "Definite Clauses" (van Emden, 1975). The basic computational mechanism of Prolog, and therefore of ERL, is a pattern matching process ('unification') operating on general record structures ('terms' of logic).*

Prolog was initially developed at the University of Marseilles (Roussel 1975) as a practical tool for 'logic programming' (Kowalski 1974; Colmerauer 1975; van Emden 1975), and has since been used in several other centers (Stanford, Edinbcrough) for writing language analysis systems (Dahl 1977; Warren 1977a, Warren 1977b).

Prolog is a perspicuous and powerful language for the expression of the concepts of our Event Representation Language, and admits of an effective and reasonably efficient implementation. Clear, readable, concise programs can be written quickly and with few errors. Specifically, the following features make it particularly suitable for our purposes:

- Pattern matching (unification) replaces the conventional use of selector and constructor functions for operating on structured data.
- The arguments of a procedure can serve, not only for it to receive one or more values as input, but also for it to return one or more values as output. Procedures can thus be "multi-output" as well as "multi-input".
- The input and output arguments of a procedure do not have to be distinguished in advance, but may vary from one call to another. Procedures can thus be "multi-purpose".
- Procedures may generate (via backtracking, in the case of Prolog) a set of alternative results. Such procedures are called "non-determinate". Backtracking amounts to a high-level form of iteration.
- Procedures may return "incomplete" results, i.e., the term or terms returned as the result of a procedure may contain variables, which are only filled in later, by calls to other procedures. The effect is similar to the use of assignment in a conventional language to fill in fields of a data structure. Note, however, that there may be many occurrences of an instantiated variable, and that all of these get filled in simultaneously (in a single step) when the variable is finally instantiated. Note also that when two variables are unified together, they become identified as one. The effect is as though an invisible pointer, or reference, linked one variable to the other. We refer to these related phenomena as the "logical variable".
- "Program" and "data" are identical in form. A procedure consisting solely of unit clauses is closer to an array, or table of data, in a conventional language.

---

* For a full description of the syntax and semantics of Prolog, see Appendix B.

Section 3.2 shows how Prolog is used for encoding templates and their associated procedures. Section 3.3 explains the ERL procedures so far developed for event record synthesis, while section 3.4 offers a detailed example of how templates written in ERL, and executed as a Prolog program behave as a semantic interpreter for the syntactic structures output by the ATN parser.

3.2 How Templates are Expressed in Prolog

In the following paragraphs we describe the formalism used for the abstract specification of both the data structures (templates) and the procedures of ERL, and show how they are expressed in the programming language Prolog.

In ERL, both templates and template slots are encoded as Prolog procedures. A Prolog procedure consists of a sequence of statements called *clauses*. A clause comprises a *head* and a *body*. The head corresponds to a procedure call, while the body represents conditions to be fulfilled for the head to be satisfied. The general format of a Prolog clause is as follows:

- head:- body.

The head consists of one Prolog *goal*, while the body may consist of a sequence of one or more such goals. Goals correspond to Prolog *terms*, which have the general form:

- functor(arg1, arg2,....argn)

Templates are encoded as 'construct' clauses. For example, the DEPLOY template, which is informally represented in Table 3-1 in a simplified form, is encoded as in Table 3-2.

Table 3-1. Informal Description of the DEPLOY Concept

Descriptive Elements			Procedural Elements
Descriptor	Filler Specification	OBL/ / /OPT	Procedures for filling slots
Object	Logical Subject noun phrase (+acft)	OBL	Construct 'aircraft' template from logical subject
Destination	PP: 'to'+ NP(+loc)	OBL	Search VMODS list for appropriate prepositional phrase
Time	1. Adv(+ time + ref) 2. PP (during, between) + NP(+ time)	OPT	Search VMODS list for appropriate constituent

Table 3-2. ERL Representation of DEPLOY Template

```
+--------------------------------------------------------------------+
| construct('DEPLOY', s(Subj,Vbgr,Obj,Compl,Vmods), [OB1,S1,L2,DTG]):- |
|                objectl (Subj,OB1),                                  |
|                destination1(Vmods,D1),                              |
|                construct('DTG',Vmods,DTG).                          |
|                                                                     |
+--------------------------------------------------------------------+
```

The head of the "construct" clause has three arguments: a template name, the name of the syntactic constituent which serves as the context which is searched in an attempt to find fillers for the descriptor slots of the template in question, and a third argument which represents the output of the procedure, i.e., the instantiated slots.

The body of the 'construct' clause consists of three 'goals' corresponding to the three slots of the DEPLOY template shown in Figure 3-2. These three goals are themselves defined as procedures, which seek fillers for the descriptor slots they represent.

Procedures for filling template slots can be obligatory or optional, and are named after the slot they are designed to fill. Thus, the procedure for filling a 'destination' slot is called 'destination1', if it obligatory, and 'destination2', if it optional. Slot-filling procedures take as input a syntactic structure and return a "filler", provided certain conditions expressed as goals are satisfied.

For example, the 'destination1' slot in the 'construct' procedure for DEPLOY, is written as in Table 3-3.

Table 3-3. A 'destination' Clause

```
+--------------------------------------------------------+
|                                                        |
|   destination(Vmods, slot('DESTINATION*',Slot)):-      |
|                fill-slot(Vmods, ['TO'],'LOC',Slot).    |
|                                                        |
+--------------------------------------------------------*
```

The 'destination' clause takes the Vmods list as input, and returns a filler which must be a prepositional phrase with a preposition 'to', and an object nounphrase with the feature 'LOC'. Notice that the third goal of the construct procedure for 'DEPLOY' is a call to the 'construct' procedure for building a DTG record. For a full listing of the procedures for constructing templates, see Subsection 3.3.

It would appear that the procedures required for filling descriptor slots will cover a wide spectrum, from those involving a straight forward match of two structures, to those requiring complex operations such as data base searches. A preliminary specification of procedures developed for the analysis of the air activities sublanguage is given in the next subsection.

3.3 ERL Procedures for Event Record Synthesis

In this section we present the set of 'air activities' event templates and their associated procedures as expressed in ERL, the formalism from which they will be compiled.

The templates developed so far for the air activities domain cover four types of entities: events, objects, relations, and concepts related to time and date (the DTG concept).

Templates have been developed for the following event classes: 'active', 'arrive', 'depart', 'deploy', 'enroute', 'flight', 'locate', 'penetrate', 'recover', and 'return'. The only physical object currently associated with a template is the 'aircraft'. How relations are encoded in Prolog is illustrated by the 'precede' template. A special template has been developed for the date time group concept (DTG).

In addition to the procedures for expressing templates, there are a number of other procedures which serve the purpose of initiating the event synthesis process, by identifying the template required for the interpretation of a particular input string. This is the function of the 'build_ER' procedure described below.

3.3.1 The 'build+ER' Procedure A 'build_ER' clause takes as input a parse tree (or a substructure thereof), finds the name of the template to be activated, invokes the corresponding 'construct' clause, and returns an event record (ER). The 'build_ER' procedure has three entry points corresponding to the three cases listed below.

3.3.1.1 Input is a Sentence:-

```
build_ER (s(Subj,Vbgr,Obj,Compl,Vmods),temp(Name,ER)):-
          find_t_name(Vbgr,Name),
          construct(Name,s(Subj,Vbgr,Obj,Compl,Vmods),ER).
```

3.3.1.2 Input is a Nominalized Sentence:-

```
build_ER(np(Det,L1,N(W,_),L2),ER):-
          feat(W,'NOMZ'),
          change(np(Det,L1,N(W,_),L2),T1),
          build_ER(T1,ER).
```

3.3.1.3 Input is a Nounphrase:-

```
build_ER(np(Det,L1,Noun,L2),ER):-
      find_t_name(Noun,Name),
      construct(Name,np(Det,L1,Noun,L2),ER).
```

'build_ER' clauses have two or more subgoals. The task of the 7find_t_name' procedure is to identify the name of the template required for the interpretation of the input, while the purpose of the 'construct' procedure is to fill in the slots of the template thus identified, i.e., to construct an event record (ER). This event record is the content representation of the input. The 'feat' clause is a built-in procedure, also referred to in Prolog as an 'evaluable predicate'. 'feat' checks a lexical entry for a given feature. For example, in 3.2.1 above, it checks the headnoun of the nounphrase np for the feature 'NOMZ'. Finally, 'change' is a normalization procedure which restores sentential structure to nominalizations.

3.3.2 The 'construct' Procedure A 'construct' clause, when activated, generates a set of subgoals which seek suitable 'fillers' for the slots of the template it embodies. The output is a list of instantiated slots which reflect the meaning content of the input. In this sense, 'construct' clauses may be regarded as procedural definitions of templates. 'Construct' clauses currently handle four kinds of entities: events (e.g., deploy), physical objects (e.g., aircraft), relations (e.g., precede), and abstract concepts such as those that pertain to date and time indications.

3.3.2.1 Construct Clauses Embodying Event Templates

3.3.2.1.1 'active'

```
construct('ACTIVE',s(Subj,Vbgr,Obj,Compl,Vmods),
                                  [OB1,MI,L1,ALT,ST2,DTG]):-
        object1(Subj, OB1),
        mission(s(_,_,Obj,Compl,Vmods),MI),
        location1 (s(_,_,Obj,_,Vmods), L1),
        altitude(IT, ALT),
        status2 (IT, ST2),
        construct('DTG',Vmods,DTG).
```

3.3.2.1.2 'arrive'

```
construct('ARRIVE',s(Subj,Vbgr,Obj,Compl,Vmods), [OB1,D2,DTG]):-
      object1 (Subj, OB1),
      destination2 (Vmods,D2),
      construct('DTG',Vmods,DTG).
```

3.3.2.1.3 'depart'

```
construct('DEPART',s(Subj,Vbgr,Obj,Compl,Vmods),[OB1,S1,L2,DTG]):-
      object1 (Subj, OB1),
      source1 (Vmods, S1),
      location2 (s(_,_,Obj,_,Vmods), L2),
      construct('DTG',Vmods,DTG).
```

3.3.2.1.4 'deploy'

```
construct('DEPLOY',s(Subj,Vbgr,Obj,Compl,Vmods), [OB1,D1,DTG]):-
      object1 (Subj, OB1),
      destination1 (Vmods,D1),
      construct('DTG',Vmods,DTG).
```

3.3.2.1.5 'enroute'

```
construct('ENROUTE',s(Subj,Vbgr,Obj,Compl,Vmods),[OB1,MI,D1,DTG]):-
      object1 (Subj, OB1),
      mission(s(_,_,Obj,Compl,Vmods),MI),
      destination1 (Vmods,D1),
      construct('DTG',Vmods,DTG).
```

3.3.2.1.6 'flight'

```
construct('FLIGHT',s(Subj,Vbgr,Obj,Compl,Vmods),
                [OB1,MI,L2,S2,D2,E2,DIR,ALT,PA,THM,DTG]):-
      object1 (Subj, OB1),
      mission(s(_,_,Obj,Compl,Vmods), MI),
      location2 (s(_,_,Obj,_,Vmods),L2),
      source2 (Vmods,S2),
      destination2 (Vmods,D2),
      extent2 (IT, E2),
      direction(Vmods, DIR),
      altitude(IT, ALT),
      path(Vmods, PA),
      them(Vmods, THM),
      construct('DTG',Vmods,DTG).
```

3.3.2.1.7 *'locate'*

```
construct('LOCATE',s(Subj,Vbgr,Obj,Compl,Vmods),[AG2,OB1,L1,DTG]):-
      agent2 (IT, AG2),
      object1 (Subj, OB1)
      location1 (s(_,_,Obj,_,Vmods), L1),
      construct('DTG',Vmods,DTG).
```

3.3.2.1.8 *'penetrate'*

```
construct ('PENETRATE',s(Subj,Vbgr,Obj,Compl,Vmods),[OB1,L1,ALT,DTG]):-
      object1 (Subj, OB1),
      location1 (s(_,_,Obj,_,Vmods), L1),
      altitude(IT, ALT),
      construct('DTG',Vmods,DTG).
```

3.3.2.1.9 *The'precede'*

```
construct('PRECEDE', s(Subj,_,Obj,_,_),[E1,E2]):-
        build_ER(Subj,E1),
        build_ER(Obj, E2).
```

3.3.2.1.10 *'recover'*

```
construct( 'RECOVER', s(Subj,Vbgr,Obj,Compl,Vmods),[OB1, L1,DTG]):-
      object1 (Subj, OB1),
      location1(s(_,_,Obj,_,Vmods),L1),
      construct('DTG',Vmods,DTG).
```

3.3.2.1.11 *'return'*

```
construct('RETURN',s(Subj,Vbgr,Obj,Compl,Vmods),
                        [OB1,D1,MI,L2,S2,DTG]):-
      object1 (Subj, OB1),
      destination1 (Vmods,D1),
      mission(s(_,_,Obj,Compl,Vmods), MI),
      location2 (s(_,_,Obj,_,Vmods), L2),
      source2 (Vmods, S2),
      construct('DTG,Vmods,DTG).
```

3.3.2.2 Construct Clauses for Templates Representing Physical Objects

3.3.2.2.1 'aircraft' Note that 'aircraft' is the only template representing a physical object in the system at present.

```
construct('AIRCRAFT',np(Det,L1,Head,L2),[EQ,NA,SUB,SB,SET]):-
          equipment(L1,Head,EQ),
          nationality(L1,'nation',NA),
          subordination(L2,SUB),
          stagingbase(L2,SB),
          setspec(Det,SET).
```

3.3.2.3 Construct Clauses Relating to Date and Time Concepts

```
construct('DTG',Vmods,[TI,DT]):-
           time(Vmods, TI),
           date(Vmods,DT).
```

3.3.3 Procedures for Filling in Template Slots The predicates used for filling template slots are represented by slot names. A slot name followed by '1' means that filling it is obligatory; a slot name followed by '2' means that filling it is optional.

3.3.3.1 *'altitude'*

```
altitude(s(_,_,_,_,Vmods),slot('ALT=', Slot)):-
                        fill_slot(Vmods, ['at'], 'ALT', Slot).

altitude(_,nil).
```

3.3.3.2 *'date'*

```
date(Vmods,slot('DATE=',L,W,Day,Month,Year)):-
                      member(pp(L,W,date(Day,Month,Year)),Vmods).
date(_, nil).
```

3.3.3.3 *'destination'*

```
destination1(Vmods,slot('DESTINATION=',Slot)):-
                      fill_slot(Vmods,['TO'],'LOC',Slot).

destination2 (X,Y):-  destination1(X,Y).

destination2 (_,nil).
```

3.3.3.4 *'direction'*

```
direction(Vmods,slot('DIRECTION=',Slot)):-
                          fill_slot(Vmods, 'DIR', Slot ).

direction(_,nil).
```

3.3.3.5 *'equipment'*

```
\ equipment(List,nnode(W,_),slot('EQUIPMENT=',[List,W])):- feat(W,'ACRAFT').
```

3.3.3.6 *'location'*

```
location1(Input,slot('LOCATION=',X)):- locat1(Input,X).
location2(Input,slot('LOCATION=',X)):- locat2(Input,X).
locat1(Input,cons(X,List)):-  loc(Input,X), locat2(Input,List).
locat2(Input,cons(X,List)):-  loc(Input,X), locat2(Input,List).
locat2(_,nil).
loc(s(_,_,_,_,Vmods), Slot):-
        fill_slot(Vmods,['ALONG','AT','EAST OF','IN','OVER'],'LOC',Slot).
loc(s(_,_,NP,_,_),NP):-  test_nhead(NP,'LOC').
```

3.3.3.7 *'mission'*

```
mission(s(_,_,_,_,Vmods),slot('MISSION=',  Slot)):-
        fill_slot(Vmods, ['AFTER', 'FROM','IN','ON'], 'ACTY', Slot).
mission(s(_,_,NP,_,_),slot('MISSION=',NP)):- test_nhead(NP,'NOMZ').
mission(_,nil).
```

3.3.3.8 *'nationality'*

```
nationality(List, Feature,slot('NATIONALITY=', W)):- member(nnode(W,_),List),
                                                      feat(W,Feature).
nationality(List, Feature,slot('NATIONALITY=',W)):- member(W,List),
                                                      feat(W,Feature).
nationality(_,_,nil).
```

3.3.3.9 *'object'*

```
object1 (NP,slot('OBJECT=', Slot)):-   test_nhead(NP,'ACRAFT'),
                                       construct('AIRCRAFT',NP, Slot).
```

3.3.3.10 *'path'*

```
path(Vmods,slot('PATH=', Slot)):-  fill_slot(Vmods,['VIA'],'LOC',Slot).
path(_,nil).
```

3.3.3.11 *'setspec'*

```
setspec(dp(_,_,Num),slot('NUMBER=',Num).
setspec(_, nil).
```

3.3.3.12 *'source'*

```
source1(Vmods,slot('SOURCE=', Slot):- fill_slot(Vmods,['FROM'],'LOC',Slot).
source2(X,Y):- source1(X,Y).
source2 (_,nil).
```

3.3.3.13 *'stagingbase'*

```
stagingbase(List,slot('STAGINGBASE=', Slot):-  fill_slot(List,['AT'],
                                               'LOC',Slot).
stagingbase(_,nil).
```

3.3.3.14 *'subordination'*

```
subordination(List,slot(SUBORDINATION='),Slot)):-
            fill_slot(List,['FROM'],'SUBNUM',Slot).
subordination(_,nil).
```

3.3.3.15 'them' (the threat)

```
them(Vmods,slot('THEM=', Slot):-
            fill_slot(Vmods, ['AGAINST'], 'NATION', Slot).
them(_,nil).
```

3.3.3.16 'time'

```
time(Vmods,slot('TIME=', Slot):-
         find_time(Vmods,['AT','BETWEEN','BY','DURING','SINCE'],'TYME',
                  Slot).
time(Vmods,slot('TIME=',Slot):-
         find_time(Vmods,['AT','BETWEEN','BY','DURING','SINCE'],'4DIG',
                 Slot).
time(Vmods,slot('TIME=',Slot):-
         fill_slot(Vmods,['AT','BETWEEN','BY','DURING','SINCE'],'TYME',
                 Slot).
time(Vmods,slot('TIME=', Slot):-
         fill_slot(Vmods, 'TYME',Slot).

time(_,nil).
```

3.3.4 Other Procedures

3.3.4.1 'fill+slot'

```
fill_slot(List, Preplist, Feature,[L1,Prep,NP]):-
                member (pp(L1,Prep,NP),List),
                member(Prepa, Preplist),lexeq(Prep,Prepa),
                        test_nhead(NP, Feature).
```

Given the Vmods list, a list of prepositions Preplist, and a lexical feature Feature, 'fill_slot' searches the Vmods list for a prepositional phrase (pp), such that Prep is a member of Preplist and the headnoun of NP has the feature Feature. 'fill_slot' returns the prepositional phrase 'pp'.

```
fill_slot(List, Feature,W):-
         member(Wa, List),lexeq(W,Wa),
         feat(W,'ADVB'),
         feat(W, Feature).
```

Given the Vmods list and a lexical feature Feature, fill_slot' searches the Vmods list for an adverb with feature Feature,and returns the adverb.

```
fill_slot(NP, Feature,NP):-  test_nhead(NP,'LOC').
```

3.3.4.2 'find+feat'

```
find_feat(W,L,Y):-
     member(Y,L),
       feat(W,Y).
```

'find_feat' takes as arguments the dictionary entry of a word W, a list of atoms naming templates available in the system (L), and returns a value for the variable Y, such that Y is a member of L, and Y is a feature of W.

3.3.4.3 'find+t+name' 'Find_t_name' is a procedure for finding the name of the template to be activated for the interpretation of a particular input structure. 'find_t_name' has two entry points according to whether the template name sought is derivable from a verbgroup or from a noun .

3.3.4.3.1 The template name is derivable from a verbgroup:-

```
find_t_name(vg(_,_,_,W),Name):-
      find_feat(W,['ARRIVE',DEPART',DEPLOY','ENROUTE','FLIGHT',
               'LOCATE','PENETRATE','PRECEDE', RECOVER',
               'RETURN'],Name).
```

3.3.4.3.2 The template name is derivable from a noun:-

```
find_t_name(nnode(W,_),Name):-
               find_feat(W,['AIRCRAFT'],Name).
```

3.3.4.4 'find+time'

```
find_time(List, Preplist, Feature,[L1,W,L2]):-
               member(pp(L1,W,L2),List),
               member(Wa,Preplist), lexeq(W,Wa),
               member(X,L2),
               feat(X,Feature).
```

3.3.4.5 'test+nhead'

```
test_nhead(np(_,_,nnode(W,_),_),Feature):- feat(W,Feature).
```

'test_nhead' determines whether the head noun (W) of the input np the feature Feature.

3.3.4.6 Listdefinition

```
list([]).
list(X,L):- list(L).
```

3.3.4.7 Listmembership

```
member(X,[X,.._]).
member(X,[_,..L]):-  member(X,L).
```

3.3.5 Syntactic Normalization Rules.

3.3.5.1 Nominalizations. The rules listed below apply to nominalizations in subject position and/or nominalizations in object position.

3.3.5.1.1 Restructuring 'Passive' Nounphrases.

Example: A WEATHER RECONNAISSANCE FLIGHT BY ONE
      PRETORIA BASED SP-256 B-80 (BEACON)
      TO THE CAPE VERDE ISLANDS.

```
change(np(Det,[L1,X], nnode(W,0),[X1,pp(_,by,Y),X2]),
          s(Y,vg(_,_,_,W),np(Det,L1,nnode(X,0),[]),_,[X1,X2])):-
                              test_nhead(Y,'NOMZ').
```

3.3.5.1.2 *Restructuring 'Active' Nounphrases*

Example1: UAF B-75 DEPLOYMENTS TO MAURITIUS

```
change(np(Det,[L1,X],nnode(W,0),L2),
     s(np(Det,L1,nnode(X,0),[]),vg(_,_,_,W),0,0,L2)).
```

Example2: DEPLOYMENT OF 12 AIRCRAFT TO KIGALI

```
change(np(Det1,L1, nnode(W1,pp(_,of,np(Det2, L2,nnode(W2,0),[]))), L3),
               s(np(0,[L2],nnode(W2,0),[]),vg(_,_,_,W1),0,0,L3)):-
                                    feat(W2,'acraft').
```

3.4 Event Record Synthesis, an Example

Before presenting an example of how templates are executed by ERL, a word should be said about the control mechanism employed by the system.

3.4.1 The ERL Control Mechanism. Prolog provides a remarkably simple form of control, which suffices for many practical applications.

The declarative semantics of Prolog clauses is such that the order of the goals in a clause and the order of the clauses themselves are both irrelevant to the declarative interpretation. However, these orderings are generally significant in Prolog, as they constitute the main control information.

When the Prolog system is executing a procedure call, the clause ordering determines the order in which the different entry points of the procedure are tried. The goal ordering fixes the order in which the procedure calls in a clause are executed. The 'productive' effect of a Prolog computation arises from the process of 'matching' a procedure call against a procedure entry point.

3.4.2 Step by Step Description of the Synthesis Process. In this section we describe by means of an example how ERL template representations drive event record synthesis. Consider the following example:

(1) THIS AIRCRAFT ROUTINELY PRECEDES UAF B-75 DEPLOYMENTS TO MAURITIUS.

As pointed out previously, one of the basic principles underlying our approach to the content analysis of narrative text is that the structural descriptions at all levels of analysis should be homogeneous. Sentence (1) above was chosen precisely because it allows us to show how the same formalism lends itself naturally to the description of structures and processes at several levels of grammatical description thus providing a homogeneous approach to the interpretation of the syntactic structures output by the ATN. Specifically, the levels of grammatical description involved in the analysis of (1) are:-

- syntactic normalization;
- the description of objects (aircraft);
- the description of an atomic event ('deployments');
- the description of a text-level relation ('precede').

Sentence (1) states that certain deployments are routinely preceded by a certain flight. Notice that syntactically, (1) is a simple sentence of the form Subject, Verb, Object. Conceptually, however, it is a complex structure in which the main verb 'precede' functions as a text-level relation locating two events on the time line. The two events are linguistically encoded as the subject and the object of the verb 'precede'. Note that the subject is 'this aircraft' which, although syntactically a simple noun phrase describing an object, is understood as 'the flight of this aircraft', i.e., it is understood as the description of an event. This is information which does not reside in the actual text, and which will eventually be supplied by an inferential component utilizing extralinguistic knowledge stored in the system. The current version of ERL lacks the necessary inferential mechanisms which would supply this information. 'This aircraft', therefore, is interpreted as the description of an object. As mentioned above, 'precede' relates two events on the time axis. 'Precede', then, is a relation which has two arguments: a 'predecessor' and a 'successor'. As indicated above, the first argument of 'precede' -- the 'predecessor'-- will be an aircraft description. The second argument of 'precede'-- the 'successor' -- will be the interpretation of the syntactic object of the sentence. ERL utilizes a normalization rule to transform the latter into a sentential structure which is then further interpreted by rules of semantic interpretation, and transformed into an event record of type 'deploy'.

A diagrammatic representation of the final output of the event record synthesizer|is given in Figure 1, which is read as follows:-

The record is of type 'precede'. The 'predecessor' describes an object of type 'aircraft', while the 'successor' describes an event of type 'deploy'. The objects being deployed are UAF B-75s, and the destination of these aircraft is Mauritius.

```
 ______________________________________________________________
| Precede                     ____________________________      |
| Modifier: ROUTINELY        |Aircraft                    |     |
| Predecessor: -------->     |Equipment: THIS AIRCRAFT    |     |
|                            |____________________________|     |
|                                                               |
|                             __________________________________|
|                            |Deploy     __________________  |  |
|                            |Object:-->|Aircraft          | |  |
|                            |          |Equipment:  B-75  | |  |
| Successor: ---------->     |          |Service:    UAF   | |  |
|                            |          |__________________| |  |
|                            |Destination:  TO MAURITIUS      |  |
|                            |                                |  |
|                            |________________________________|  |
|                                                               |
|_______________________________________________________________|
```

Figure 1. Content Representation of "THIS AIRCRAFT ROUTINELY PRECEDES UAF B-75 DEPLOYMENTS TO MAURITIUS".

3.4.2.1 The Initiation of the Synthesis Process. In this section we give a detailed step by step description of the event record synthesis process as executed by MATRES II. As explained in a previous section, the ERL semantic interpretation rules (clauses) are used top-down, one at a time. Goals in a clause are executed from left to right. If there are alternative clauses at any point, backtracking will return to them. To see how parse trees are interpreted by ERL, consider (2), which is the parse tree of sentence (1):-

```
              (2) s(np(dp(0, THIS,0),[], nnode(AIRCRAFT,0),[] ),
                       vg([ROUTINELY],[],0,PRECEDES),
                      np(0,[nnode(UAF,0),nnode(B-75,0)],
                             nnode(DEPLOYMENTS,0),
                   [pp([],TO,np(0,[],nnode(MAURITIUS,0),[]))]),
                                    0,[]).
```

For simplicity of exposition we will henceforth refer to structure (2) as 'Tree_in'.

The synthesis process involves the execution of the system-generated goal (3):

```
  (3)  :- build_ER ('Tree_in' ,ER).
```

'build_ER' clauses have two arguments: the input structure 'Tree_in', which in our case is the structure given in (2), and an output structure ER, which is the content representation of 'Tree_in'.

3.4.2.2 *Activation of Template.* Since 'Tree_in' in our example is a sentential structure, goal (3) unifies with the head of the first clause of the 'build_ER' procedure (4):

```
  (4) build_ER (s(Subj,Vbgr,Obj,Compl,Vmods),ER):-
       find_t_name(Vbgr,Name),
       construct(Name,'Tree_in' ,ER).
```

This results in the following instantiations:

```
  (5) Subj = np(dp(0,THIS,0),[],nnode(AIRCRAFT,0),[]);
      Vbgr = vg([ROUTINELY],[],0,PRECEDES);
      Obj  = np(0,[nnode(UAF ,0),nnode(B-75,0)],
                  nnode(DEPLOYMENTS,0),
                [pp([],TO,np(0,[],nnode(MAURITIUS,0),[]))]);
      Compl = 0;
      Vmods = [].
```

The body of the matching clause instance (4) also gives rise to the two new subgoals (6) and (7):

```
  (6) find_t_name(vg([ROUTINELY],[],0,PRECEDES),Name).

  (7) construct(Name,
            s(np(dp(0,THIS,0),[],nnode(AIRCRAFT,0),[]),
            vg([ROUTINELY] ,[],0,PRECEDES),
            np(0,[nnode(UAF ,0),nnode(B-75,0)],
                 nnode(DEPLOYMENTS,0),
              [pp([],TO,np(0,[],nnode(MAURITIUS,0),[]))]),
            0,[]),ER).
```

The first task is to identify the template required for the interpretation of (2). This is achieved by executing goal (6) listed above.

Goal (6) matches the head of the first clause of the 'find_t_name' procedure (see 8). It produces the instantiations in (9), and yields the new goal (10):-

```
(8) find_t_name(vg(_,_,_,W),Y):-
        find_feat(W,L,Y).
```

```
(9) W ='precedes' ; Y = Name
```

```
(10) find_feat ('precedes', [list of event template names], Name).
```

Goal (10) in turn unifies with the head of the 'find_feat' clause (11)

```
(11) find_feat (W,L,Y):-
         mem (Y,L),
         feat (W,Y).
```

This creates the following instantiation (12):-

```
(12) find_feat ('precedes', [list of event template names], Name):-
         mem(Name,[list of event template names]),
         feat('precedes', Name).
```

The execution of the subgoals of (12) result in the instantiations (13):-

```
(13.1) Name = 'Precede' , and
(13.2) construct('precede', Tree-in', ER).
     where (13.2) is still only a partial instantiation of (7).
```

Goal (6) is now fully instantiated, i.e., the name of the template sought was found to be 'precede'.The system now proceedes to execute second goal set up by executing (3), namely goal (7), now instantiated to (13.2). Executing this goal results in the instantiation of the two arguments of 'precede', namely, E1 and E2.

3.4.2.3 Instantiating the Arguments of 'PRECEDE' The reader is reminded that the verb 'precede' is a two-place predicate whose interpretation in the environment of a subject E1 and an object E2 is 'before(E1,E2)'. The 'construct' procedure for 'precede' seeks to find fillers for the two arguments E1 and E2. To achieve this result, goal (13.2) unifies with the head of the 'contsruct' clause for 'precede' (14), and sets up the two subgoals (14.1) and (14.2):-

```
(14)    construct ('precede', s(Subj,_,Obj,_,_), [E1,E2]):-
(14.1)                  build_ER(Subj,E1),
(14.2)                  build_ER(Obj, E2).
```

where, according to (5),

```
Subj=np (dp(0,THIS,0),[],nnode(AIRCRAFT,0),[]);
Obj=np(0,[nnode(UAF,0),nnode(B-75,0)],
     nnode(DEPLOYMENTS,0),
     [pp([],TO,np(0,[],nnode(MAURITIUS,0),[]))]).
```

The next step is to execute goals (14.1) and (14.2).

3.4.2.4 Interpreting the Syntactic Subject. The partially instantiated goal (14.1) is shown in (15):-

```
(15) build_ER(np(dp(0,THIS,0),[],nnode(AIRCRAFT),0),[]),ER).
```

Since the first argument of (15) is a nounphrase, it will unify with the head of the second 'build_ER' clause (16):-

```
(16) build_ER(np(Det,L1,N(W,_),L2),ER):-
        feat(W,'NOMZ'),
        change(np(Det,L1,N(W,_),L2),T1),
        build_ER(T1,ER).
```

However, the first goal of clause (16) requires that the headnoun have the feature 'NOMZ'. This is not the case in our example, so that the first goal fails. The system now backtracks, i.e., it rejects the most recently activated clause (16) undoing any substitutions made by the match with the head of the clause. Next, it reconsiders the original goal (15) which activated the rejected clause, and tries to find a subsequent clause which also matches the goal. As a result, goal (15) now unifies with the head of the third 'build_ER' clause (17):-

```
(17)   build_ER(np(Det,L1,Noun,L2),ER):-
(17.1)     find_t_name(Noun,Name),
(17.2)     construct(Name,np(Det,L1,Noun,L2),ER).
```

This results in the following instantiations:-

```
(18) Det = dp(0,THIS,0);
     L1 = [];
     Noun = nnode(AIRCRAFT,0);
     L2 = [];
     E1=ER.
```

The first goal of (17) unifies with (19):-

```
(19) find_t_name(nnode(W,0),Y):-
        find_feat(W,['aircraft', 'DTG', etc], Y).
```

The procedure here is similar to that described earlier. As a result of the unification process, and of executing (19), we have the following instantiation:-

```
W = 'AIRCRAFT'
Y = Name = 'aircraft'.
```

Clause (17.1) is now fully instantiated -- the template sought has been found to be the 'aircraft' template. The system proceedes to the execution of goal (17.2).

Goal (17.2) is now partially instantiated to (20):-

```
(20) construct('aircraft', np(0,THIS,0),[],
          nnode(AIRCRAFT,0),[]),ER).
```

Goal (20) activates the 'construct' procedure for 'aircraft', which fills the 'equipment' slot with 'this aircraft', and leaves all other slots empty. The result of executing (20) is:-

```
       +------------------------------+
       | aircraft                     |
E1  =  | equipment= THIS AIRCRAFT     |
       |                              |
       +------------------------------+
```

3.4.2.5 Interpreting the Syntactic Object. Rather than describing the process of synthesizing a record for 'this aircraft' in detail, we will return to the second goal of the 'construct' clause for 'precede', namely, to (14.2), which is now partly instantiated to (21):-

```
(21) build_ER(np(0,[nnode(UAF,0), nnode(B-75,0)],
               nnode(DEPLOYMENTS,0),
               [pp([],TO,np(0,[],nnode(MAURITIUS,0),[]))]),ER).
```

The first argument of this clause is the nominalized sentence 'UAF B-75 DEPLOYMENTS TO MAURITIUS'.Accordingly, clause (21) will unify with the head of the second 'build_ER' clause, namely (16), reproduced here as (22) in its partly instantiated form, complete with its subgoals (22.1), (22.2), and (22.3):-

```
(22)   build_ER(np(0,[nnode(UAF,0),nnode(B-75,0)],
               nnode(DEPLOYMENTS,0),
               [pp([],TO,np(0,[],nnode(MAURITIUS,0),[]))]),ER):-
(22.1)       feat(DEPLOYMENTS, 'NOMZ'),
(22.2)       change(np(0,[nnode(UAF,0),nnode(B-75,0)],
                nnode(DEPLOYMENTS,0),
                [pp([],TO,np(0,[],nnode(MAURITIUS,0),[]))]),T1),
(22.3)       build_ER(T1,ER).
```

Goal (22.1) succeeds, and the system activates the 'change' procedure. Goal (22.2) unifies with (23) below, which restructures the input nounphrase into a sentential structure:-

```
(23)   change(np(Det,[L1,X],nnode(W,0),L2),
           s(np(Det,L1,nnode(X,0),[],),vg(_,_,0,W),0,0,L2)).
```

Upon unification with (22.2), (23) becomes instantiated to (24):

```
(24)  change(np(0,[nnode(UAF,0),nnode(B-75,0)],
             nnode(DEPLOYMENTS,0),
             [pp([],TO,np(0,[],nnode(MAURITIUS,0),[]))]),

             s(np(0,[nnode(UAF,0)],
                nnode(B-75,0),[]),
                vg([],[],0,DEPLOYMENTS),
                0,
                0,
                [pp([],TO,np(0,[],nnode(MAURITIUS,0),[]))])).
```

T1 is instantiated to the second argument of (24). The system now proceedes to execute goal (22.3) reproduced here in its instantiated form (25):-

```
(25)  build_ER(s(np(0,[nnode(UAF,0)],
               nnode(B-75,0),[],
               vg([],[],0,DEPLOYMENTS),
               0,0,
               [pp([],TO,np(0,[],nnode(MAURITIUS,0),[]))]),ER).
```

Execution of the 'built_ER' goal (25) eventually results in the activation of the 'construct' clause for 'deploy' (26):-

```
(26) construct('deploy' ,IT, [O1,D1,T2]):-
      object1 (IT, O1),
      destination1 (IT, D1),
      time2 (IT, T2).
```

with 'IT' instantiated to the first argument of (25). The goal 'object1' activates the 'object1' procedure (28):-

```
(28) object1 (s(Subj,_,_,_,_), Slot):-
        test_nhead(Subj, 'acraft'),
        construct 'aircraft', Subj, Slot).
```

The result is the instantiation:-

```
Subj = np(0,[nnode(UAF,0)],nnode(B-75,0),[]).
```

and 'Slot' gets linked to 'O1'.

The goal 'test_nhead' determines whether the headnoun(W) of a noun phrase 'np' has the feature Feature. It unifies with the clause for 'test_nhead' (30), and results in the instantiations (31):-

```
(30) test_nhead (np (_,_,nnode(W,_)),Feature
(31) W= 'B-75'; Feature = 'acraft'
```

Goal (30) succeeds, and the system begins executing the second goal of (28) namely (33):-

```
(33) construct ('aircraft', Subj,ER).
```

The second goal of (26) activates the 'destination' procedure (35) and returns D1 = 'To Mauritius'

```
(35) destination (s(_,_,_,_,Vmods), Slot):-
       fill_slot(Vmods, ['nil', 'to',....], 'loc', Slot).
```

The third goal of (26) activates the 'time2' procedure (37), which returns T2 = 'nil'.

```
(37) time2 (s(_,_,_,_,Vmods), Slot):-
        fill_slot( Vmods, ['at', 'between', 'by', 'during'],
        'tyme', Slot).

     time2 (s(_,_,_,_,Vmods), Slot):-
        fill_slot(Vmods, 'tyme', Slot).

     time2 (_,nil).
```

This completes the execution of goal (26). As a result, the second output element (E2) of the 'construct' procedure for 'precede' is instantiated to an event record of type 'deploy', i.e.,

```
       +------------------------------+
       | deploy                       |
       |                              |
       |          ___________________ |
       | object= | aircraft          | |
E2 =   |         | equipment= B-75   | |
       |         | service= UAF      | |
       |         |___________________| |
       |                              |
       | destination= TO MAURITIUS    |
       +------------------------------+
```

3.4.2.6 *Output of Event Record Synthesis Process.* The complete diagrammatic representation of the content of (1) is given in figure 1, which shows the relation between E1, E2, and its subparts.

The text-level semantic interpretation rule S1 of the Matres II System now interprets the results as follows:-

S1. ['precede', 'Tree_in', [E1, E2]] => before(E1, E2)

meaning "The content of E1 happens before the content of E2".

This completes our account of the interpretation of sentence (1).

4.0 The MATRES II System

4.1 Introduction

MATRES II is the result of the second cycle in the development of a system with full capabilities for deriving formatted records from the narrative text of intelligence messages. It represents a considerable advance on MATRES I, which provided only a rudimentary capability for testing algorithms for narrative text analysis.

The primary subject domain of MATRES II is that of air activities. While in a fully developed system the unit of analysis would be the entire message, the scope of the current system is still limited to the analysis of single sentences.

The MATRES I parser has undergone considerable refinement and expansion and currently accepts a much wider range of syntactic constructions than was previously achieved. The definition of the input language accepted by the system is embodied in a transition network grammar model based upon Woods (1970, 1973). A detailed description of the syntactic constructions accepted by the current system is given in subsection 4.3.

Since the transition network parsing methodology is by now quite well known, little will be said about the parser itself. Part II of this report, however, does include detailed documentation of our particular implementation. In this section, we focus mainly upon the parsing strategy adopted in MATRES II, including the augmented transition nets used by the system. This is the subject of subsection 4.4.

In the current system English language words are entered into a linguistic dictionary, while strings with fixed patterns are recognized at the input stage by a finite state automaton (FSA) designed especially for this purpose.

The major feature of MATRES II is its capability for semantic analysis. This is achieved by means of the Event Representation Language, which is a language specially developed for mapping the syntactic structures produced by the parser into template-derived content representations. As discussed in Section 3, the basic data structure of the Event Representation Language is the template. Section 4.5 describes the template inventory so far developed for the aircraft domain, and presents the methodology for the selection of the descriptors to be included in templates of a particular subject domain.

The next section provides a brief functional description of MATRES II.

4.2 MATRES II -- Functional Description

An overview of the MATRES II system organization and data flow is shown in Figure 4-1. The main system components are: the Lexical Unit Recognizer, the ATN parser, and the ERL "machine". The direction of the arrows in the Figure indicate the general flow of information as a sentence is processed through the system. The main stages of event record generation are shown across the center of the Figure. The analysis begins when an input sentence is received by the Lexical Unit Recognizer, which uses a stored dictionary and the FSA Recognizer to transform the individual words of an input sentence into a string of lexical units. First, a dictionary look-up process replaces words and phrases in the sentence with corresponding lexical entries. Strings which have no entries in the dictionary are passed to the FSA Recognizer, which attempts to identify them as one of several fixed-pattern categories. The output of this stage is a string of lexical units containing syntactic and semantic information for use by the parser, and later by the ERL interpretive routines.

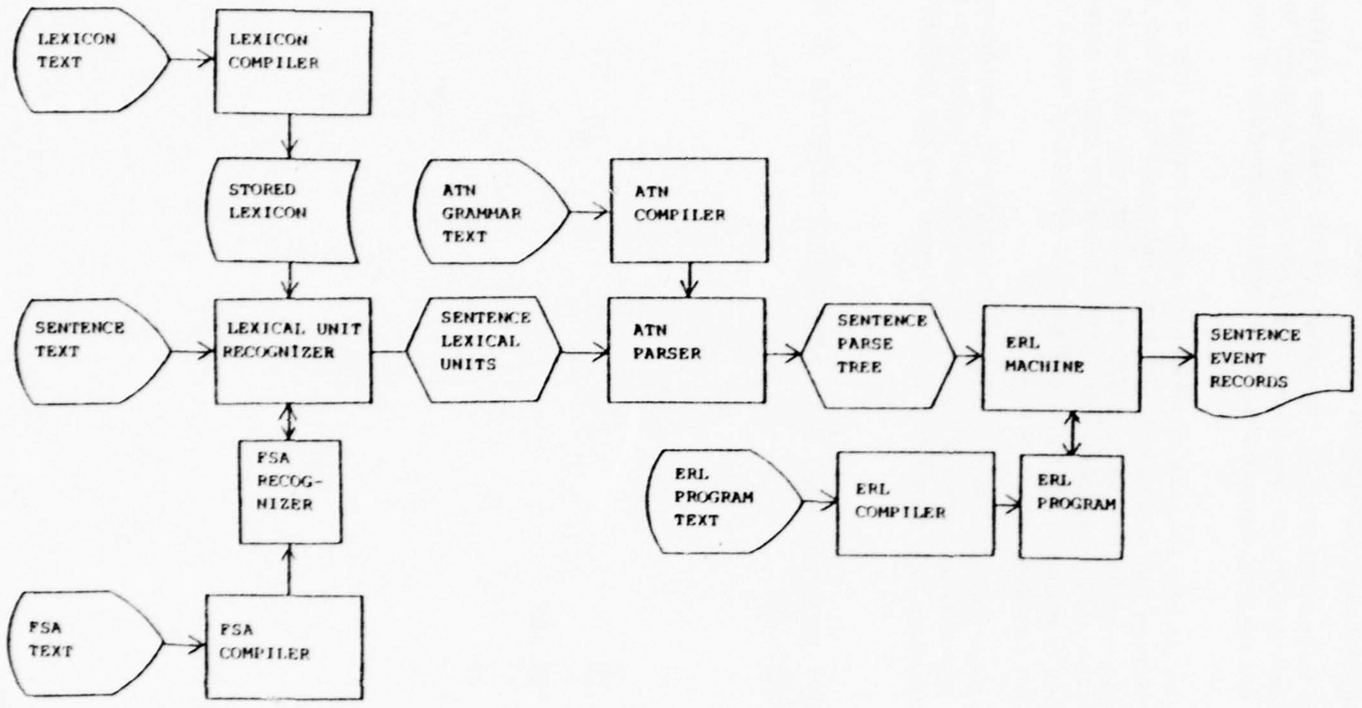


Figure 4-1. MATRES II System Overview

Next, the string is input to the parser, which analyzes it according to the sublanguage grammar stored in the system, and produces a parse tree showing the constituent structures of the input string and their hierarchical relationships.

The parse tree in turn is input to the ERL "machine", which uses the pattern matching process ("unification mechanism") of the Event Representation Language to produce a set of one or more event records representing the information content of the input sentence.

Feeding into this are the various analysis components, each compiled from a source text in a language appropriate to the component. The base language for all the programs -- except the ERL machine -- is Forth; the respective compilers are written in FORTH and the compiled form of the various components is the threaded code characteristic of Forth. The ERL compiler is coded in SPITBOL, a dialect of SNOBOL 4, which was chosen because of its excellent facilities for compiler writing.

The following are two examples showing the internal processing of sentences. The first example shows the parse tree followed by the event record produced by the ERL machine; the second example only shows the input sentence and the MATRES II output.

Example 1

```
*>> TWO UGANDAN ACFT FROM REGIMENT A313 AT ENTEBBE DEPLOYED TO GULU
*AT 0200Z ON 21 FEBRUARY.
PARSE OUTPUT:
LIST OF:
| NODE: 1|S
| | LIST OF:
| | | NODE: 2|PP
| | | | NODE: 4|DATE
| | | | | <<NIL>>
| | | | | 392.. FEBRUARY
| | | | | LIST OF:
| | | | | | 372.. 21
| | | | | END LIST
| | | | END NODE
| | | | 352.. ON
| | | | LIST OF:
| | | | END LIST
| | | END NODE
| | | NODE: 2|PP
| | | | LIST OF:
| | | | | 332.. 0200Z
| | | | END LIST
| | | | 312.. AT
| | | | LIST OF:
| | | | END LIST
| | | END NODE
| | | NODE: 2|PP
| | | | NODE: 2|NP
| | | | | LIST OF:
| | | | | END LIST
| | | | | NODE: 5|NNOD
| | | | | | | <<NIL>>
| | | | | | | 292..GULU
| | | | | | END NODE
| | | | | | LIST OF:
| | | | | | END LIST
| | | | | | <<NIL>>
| | | | | END NODE
| | | | | 272..TO
| | | | | LIST OF:
| | | | | END LIST
| | | | END NODE
| | | END LIST
| | | <<NIL>>
| | | <<NIL>>
| | | NODE: 2|VG
| | | | 232.. DEPLOYED
| | | | <<NIL>>
| | | | LIST OF:
| | | | END LIST
| | | | LIST OF:
| | | | END LIST
| | | END NODE
| | | NODE: 2|NP
| | | |  LIST OF:
| | | | | NODE: 2|PP
| | | | | | NODE: 2|NP
| | | | | | | LIST OF:
| | | | | | | END LIST
| | | | | | | NODE: 5|NNOD
| | | | | | | | <<NIL>>
| | | | | | | | 212.. ENTEBBE
| | | | | | | END NODE
| | | | | | | LIST OF:
| | | | | | | END LIST
| | | | | | | <<NIL>>
| | | | | | END NODE
| | | | | | 192.. AT
| | | | | | LIST OF:
| | | | | | END LIST
| | | | | END NODE
| | | | | NODE: 2|PP
| | | | | | NODE: 2|NP
| | | | | | | LIST OF:
| | | | | | | END LIST
| | | | | | | NODE: 5|NNOD
| | | | | | | | <<NIL>>
| | | | | | | | 172..A313
| | | | | | | END NODE
| | | | | | | LIST OF:
| | | | | | | | NODE: 5|NNOD
| | | | | | | | | <<NIL>>
| | | | | | | | | 182.. REGIMENT
| | | | | | | | END NODE
| | | | | | | END LIST
| | | | | | | <<NIL>>
| | | | | | END NODE
| | | | | | 152.. FROM
| | | | | | LIST OF:
| | | | | | END LIST
| | | | | END NODE
| | | END LIST
| | | NODE: 5|NNOD
| | | | <<NIL>>
| | | | 132..ACFT
| | | END NODE
| | | LIST OF:
| | | | 112.. UGANDAN
| | | END LIST
| | | NODE: 2|DP
| | | | | LIST OF:
| | | | | | 92.. TWO
| | | | | END LIST
| | | | | <<NIL>>
| | | | | <<NIL>>
| | | END NODE
| | END NODE
| END NODE
END LIST
Event: DEPLOY
Object:
...Equipment= UGANDAN ACFT
...Nationality= UGANDAN
...Subordination= FROM REGIMENT A313
...Stagingbase= AT ENTEBBE
...Number= TWO
Destination= TO GULU
Time= AT 0200Z
Date= ON 21 FEBRUARY
EVENT RECORD COMPLETE.
```

Example 2

```
*PRTREE OSET
*>> THE TWO ACFT WERE ENROUTE TO NAIROBI ON RECONNAISSANCE.
Event: ENROUTE
Object:
...Equipment= ACFT
...Number= TWO
Mission= ON RECONNAISSANCE
Destination= TO NAIROBI
EVENT RECORD COMPLETE.
```

4.3 Linguistic Grammar and Lexicon for Aircraft Domain

4.3.1 The Grammar. In this section we give an informal description of the major grammatical constituents which are recognized by the MATRES II parser, and of the analyses which are given them. The parser itself is described in section 4.4

4.3.1.1 The Declarative Sentence. A declarative sentence may be a simple sentence, as in (1), or it may be a simple sentence conjoined by a sentence conjunction with another simple sentence (of a special type) or with a noun phrase, as in (2) and (3).

(1) THE AIRCRAFT WERE ENROUTE HOMEBASE AT 0200Z.

(2) THE AIRCRAFT WERE ENROUTE HOMEBASE AT 0200Z AFTER
CONDUCTING A RECONNAISSANCE MISSION OVER THE RED SEA.

(3) THE AIRCRAFT WERE ENROUTE HOMEBASE AT 0200Z AFTER
A RECONNAISSANCE MISSION OVER THE RED SEA.

The MATRES II grammar analyzes a declarative sentence as a list having as its first element a simple sentence, which may be followed optionally by a sentence conjunction and either another simple sentence or a noun phrase.

4.3.1.2 The Simple Sentence. A simple sentence has a noun phrase subject followed by a verb group, optionally followed by a direct object, a complement, and one or more post-verb modifiers.

The grammar analyzes a simple sentence as a five-branched node structure. The first branch points to the subject, the second branch to the verb group, the third to the object, the fourth to a complement, and the fifth to a list of adverbial modifiers.

4.3.1.3 The Noun Phrase. A noun phrase may consist of a determiner followed by a list of pre-head modifiers, a head noun, and a list of post-head modifiers.

A determiner may consist simply of an article (eg. 'THE'), a quantifier (eg. 'ALL'), or a number phrase (eg. 'AS MANY AS SIX'), or it may be a complex structure involving two or three of these constituents, as in (4) through (7).

(4) ALL THE AIRCRAFT

(5) ALL SIX AIRCRAFT

(6) THE SIX AIRCRAFT

(7) ALL OF THE SIX AIRCRAFT

Pre-head modifiers may include adjectives, nouns, past participles, and present participles. In the aircraft domain, head nouns are typically preceded by several modifiers referring to attributes such as nationality, subordination, equipment type, etc., as in (8).

(8) RETURNING UGANDAN UBBC SR-71 AIRCRAFT

Possible post-head modifiers are relative clauses, reduced relative clauses, and prepositional phrases. An example of each is given in (9) through (11), respectively.

(9) THE AIRCRAFT WHICH WERE STAGING FROM ENTEBBE

(10) THE AIRCRAFT STAGING FROM ENTEBBE

(11) THE AIRCRAFT FROM ENTEBBE

A noun phrase is analyzed as a four-branched node. The first branch points to a determiner (possibly null, as in (8)), the second to a list of pre-head modifiers, the third to the head noun, and the fourth to a list of post-head modifiers.

As a heuristic device, we allow only simple noun phrases (i.e., those without post-head modifiers) to occur as direct objects or prepositional objects. The reason for this is best illustrated by an example. In (12), we wish to analyze the relative clause 'WHICH CONDUCTED OPERATIONS OVER THE RED SEA' as a post-head modifier of 'AIRCRAFT' rather than 'ENTEBBE'. This is effected by requiring that 'ENTEBBE', which is a prepositional object, have no post-head modifiers.

(12) THE AIRCRAFT FROM ENTEBBE WHICH CONDUCTED OPERATIONS
OVER THE RED SEA

Likewise, in the embedded sentence, by requiring that the object of 'CONDUCTED' be a simple noun phrase, we achieve the desired analysis of 'OVER THE RED SEA' as an adverbial modifier, rather than a post-head modifier of 'OPERATIONS'.

4.3.1.4 The Verb Group The verb group may consist of an auxiliary followed by a verb, as in (13), or an auxiliary followed by a copula followed by an adjective, as in (14).

(13) HAVE BEEN CONDUCTING

(14) HAVE BEEN ACTIVE

In (13) the auxiliary is 'HAVE BEEN', while in (14) the auxiliary is 'HAVE', and 'BEEN' is the copula.

Some verbs (eg. 'CONDUCT', 'PENETRATE') must be followed by a direct object constituent, which is another noun phrase. Other verbs (eg. 'ARRIVE') never have a direct object, while for others (eg. 'OPERATE') the object is optional.

Adverbial modifiers include prepositional phrases and adverbs, and may occur before the subject, as in (15), after the verb (and the object, if there is one) as in (16), or embedded within the verb group, as is the case with 'CURRENTLY' in (17).

(15) AT 0200Z ON 22 FEBRUARY, THE AIRCRAFT PENETRATED ENEMY AIRSPACE.

(16) THE AIRCRAFT FLEW NORTH OVER THE INDIAN OCEAN.

(17) THE AIRCRAFT ARE CURRENTLY ACTIVE OVER THE RED SEA.

4.3.1.5 Adverbials. Pertaining to adverbial modifiers, there are several constructions which are peculiar to our particular message domain. Principle among these are time phrases and date phrases, which, along with noun phrases, are accepted as prepositional objects. Some examples of time phrases are given in (18) through (20).

(18) 0200Z

(19) 0200-0400Z

(20) 0200Z TO 0400Z

Date phrases are analyzed as three branched nodes. The first branch points to the day, the second to the month, and the third to the year (the third is often null). Some examples are given in (21) through (23).

(21) 22 FEBRUARY

(22) 22 FEBRUARY 74

(23) THE 22ND FEBRUARY

4.3.1.6 Passive Sentences. A sentence such as (24) can be paraphrased as (25), where the logical subject becomes the grammatical object, and the logical object becomes the grammatical subject.

(24) DURING THE 0200Z HOUR, FOUR AIRCRAFT FROM RGT XB412
CONDUCTED COMMAND-AND-CONTROL OPERATIONS.

(25) DURING THE 0200Z HOUR, COMMAND-AND-CONTROL OPERATIONS
WERE CONDUCTED BY FOUR AIRCRAFT FROM RGT XB412.

The MATRES II grammar reverses the passive transformation, so that the analyses of (24) and (25) are identical, with 'FOUR AIRCRAFT FROM RGT XB412' as the subject and 'COMMAND-AND-CONTROL OPERATIONS' as the direct object.

4.3.2 The Lexicon. The MATRES II lexicon is designed to support the grammatical analysis procedure. It consists of two parts:

(i) a collection of lexical entries in the form of static declarations of lexical items and their attributes, and

(ii) a listing of the features or attributes employed by the system.

The attributes fall into several classes. Examples of each are given below.

(i) Major Grammatical Category Specifications.

ADVB (adverb)
ADJ (adjective)
ART (article)
NUM (number)
N (noun)

(ii) Lexical Features:

DIR (directional)
LOC (locational)
SUBNUM (subordination number)
TENSED (marks tensed verbs)

(iii) Event Category Features:

ARRIVE
CONTINUE
DEPART
DEPLOY

A portion of the lexicon is given in figure 4-2, and a complete listing is given in Appendix C.

```
:: FLEET [ N SG ] .;
:: FLEW [ VB TRANS TENSED DIR FLIGHT ] .;
:: FLIGHT [ N SG NOMZ ] .;
:: FLIGHTS [ N [illegible] NOMZ ] .;
:: FLOGGER [ N NATO ACRAFT ] .;
:: FODDER [ N NATO ACRAFT ] .;
:: FOLLOWING [ SCONJ ] .;
:: FOR [ PREP ] .;
:: FOUR [ NUM ] .;
:: FRESCO [ N NATO ACRAFT ] .;
:: FROM [ PREP ] .;
:: GENERAL [ ADJ ] .;
:: GROUP [ N ] .;
```

Figure 4-2. Sample Lexical Entries

4.4 The MATRES II Parser

4.4.1 General Description. The MATRES II system uses an augmented transition network (ATN) parser based on Woods (1970, 1973). The general features of ATN parsers have been discussed in detail in previous OSI reports (RADC-TR-75, RADC-TR-77-194). In this section we review a few of these features with particular attention to their implementation in the MATRES II system.

An ATN grammar consists of a finite set of states connected by labeled directed arcs. Associated with each arc is a set (possibly null) of conditions and actions. The arc represents a transition from the state at its tail to the state at its head, which may be made if the appropriate conditions are met. When such a transition is made, the actions associated with that arc are executed.

In addition to the above components, there is also a push-down store (to be explained below in connection with PSH arcs) and a set of registers, including the special register *, which usually contains the current input symbol of the string being parsed.

An input string is processed from left to right, beginning at the leftmost symbol of the string and a designated initial state. As transitions are made through the net, the input string is advanced, so that the current input symbol is in turn the first symbol of the input string, the second, and so on. A sentence is accepted when a final state, an empty push-down store, and the end of the input string are all achieved simultaneously.

At this point the reader may wish to refer to the diagrams of the ATN grammar given in figures 4-3 through 4-23, and the grammar listing given in Appendix D.

4.4.1.1 Arc Types. There are five different arc types in the MATRES II grammar. The operation of these arcs is described informally in what follows. For a formal definition of the MATRES II grammar language, see part II.

A CAT arc may be taken if the current input symbol belongs to the category (or categories) specified by the arc label (and if the condition associated with that arc is satisfied). For example, a transition via an arc labeled 'CAT [ADJ]' may be made only if the current input symbol belongs to the category ADJ (i.e., has the feature ADJ). When a transition is made via a CAT arc, the input string is advanced to the next symbol.

Transition via a WRD arc is permitted just when the current input symbol is identical to the word specified by the arc. For example, an arc labeled 'WRD " BY"' may be taken only if the current input symbol is 'BY'. The input string is then advanced to the next symbol.

A TST arc may be taken if the the condition associated with that arc is satisfied. Conditions are described in more detail in the next section, but an example should suffice to give the general idea. Transition via an arc labeled 'TST * [N] * [ADJ] OR' is permitted when the contents of the register * (the current input symbol) is a member of either the category N or the category ADJ. The input string is advanced to the next symbol.

A PSH arc transfers control to the state named by the arc label, while the state at the head of the arc is saved on the push-down store. For example, the arc ':PSH TO NP/ => S/SUBJ ,,' has the effect of transferring control to state NP/ and placing state S/SUBJ on top of the push-down store. When a POP arc is taken, control is transferred to the state at the top of the push-down store, and that state is removed from the push-down store. PSH and POP arcs do not advance the input string.

A JUMP arc permits a transition to the state named by the arc label without advancing the input string. For example, the arc ':JUMP S/OBJ ,,' transfers control to state S/OBJ with the current input symbol remaining the same.

4.4.1.2 Conditions. A condition may be used to test the contents of a register for a given feature, numerical value, or lexical item. For example, the condition '* [RELPRO]' is satisfied just when the current input symbol has the feature RELPRO, and the arc ':PSH * [RELPRO] !! TO R/ => POSTMODS/P ,,' may be taken under exactly the same circumstances. The condition 'PASSIVE GETR 1 =' is satisfied just in case the contents of PASSIVE is 1, and the condition '* " BY"' is satisfied just when the current input symbol is the lexical item 'BY'.

Conditions may be formed by combining tests of the sort described above with the Boolean operators 'AND' and 'OR'. For example, the condition 'PASSIVE GETR 1 = * " BY" AND' is satisfied just in case the contents of PASSIVE is 1 and the current input symbol is 'BY'.

4.4.1.3 Actions. The action functors 'SETR' and 'GETR' are used for filling registers and retrieving the contents of registers, respectively. For example, the action '* PREPRG SETR' stores the current input symbol in the register PREPRG.

The structure-building functors 'ADDLIST' and 'NODE' are used to build lists and nodes, respectively. For example, the action '* SENT ADDLIST' takes the contents of the register * and adds it to the list SENT. The action 'PREPRG GETR OBJ GETR PP NODE' builds a two-branched node labeled 'PP', the first branch of which points to the contents of PREPRG and the second of which points to the contents of OBJ. Notice that the functor 'GETR' is used to retrieve the contents of the registers.

In connection with PSH and POP arcs we must distinguish between "pre-actions" and "post-actions". A pre-action on a PSH arc is executed before control is transferred to the state named on the arc label, while a post-action is executed when control is "popped" to the state named at the head of the PSH arc. For example, when a transition is made via the arc ':PSH PASSIVE SENDR TO S/VG * SENT ADDLIST => DCL/S ,,', the action 'PASSIVE SENDR' (to be explained below) is executed before computation begins at state S/VG. The action '* SENT ADDLIST' is executed when control is popped to state DCL/S. It should be mentioned here as well that upon returning from a "push", the register * contains whatever structure is named by the preceding POP arc. For instance, if in the example given above control had been popped to state DCL/S by the arc ':POP SUBJ GETR VP GETR S NODE ,,', then * would contain a two-branched node labeled 'S', and this is the structure which would added to the list SENT by the post-action '* SENT ADDLIST'.

The functors 'SENDR' and 'SENDL' are used to send a value to a register or a list at a lower level of computation. For example, when a transition via the arc ':PSH VMODS SENDL TO VM/ => S/S ,,' is made, computation begins at state VM/ with the list VMODS containing exactly what it did at the state from which the push was made (normally lists and registers are empty upon entry into a lower level of computation). 'SENDR' and 'SENDL' are used only as pre-actions on PSH arcs.

The functors 'RETR' and 'RETL' are used only as post-actions on PSH arcs,e and are complementary to 'SENDR' and 'SENDL' in that they retrieve register and list values from a lower level of computation at the time of a pop from that level. Consider, for example, the arc ':PSH VMODS SENDL TO VM/ VMODS RETL => S/S ,,'. Upon popping to state S/S, VMODS contains whatever it contained before the push to VM/, in addition to whatever was added to it at the level of VM/.

4.4.2 The Parsing Strategy. In this section a series of examples is used to describe the manner in which certain grammatical constituents are processed by the MATRES II ATN parser. The following notation for output structures is employed throughout: List elements are enclosed in square brackets and separated by commas. The entities pointed to by the branches of a node are enclosed in parentheses and separated by commas, with the node label preceding the left parenthesis. Lexical units are enclosed in single quotation marks.

4.4.2.1 The Declarative Sentence. The transition network for declaratives (see Figure 4-3 and Appendix D) accepts sentences consisting of a simple sentence followed optionally by a sentence conjunction and either another simple sentence or a noun phrase. It returns a list having as its first member an S (sentence) node, which may be followed by a sentence conjunction and either another S node or an NP (noun phrase) node. For example, given (1), the declarative net returns the structure given in (2).

(1) THE AIRCRAFT WERE ENROUTE HOMEBASE AFTER
    CONDUCTING OPERATIONS OVER THE RED SEA.

(2) [S(THE AIRCRAFT WERE ENROUTE HOMEBASE),
    'AFTER',
    S(CONDUCTING OPERATIONS OVER THE RED SEA)]

(1) is parsed by the declarative net as follows:

4.4.2.1.1 State DCL/. At state DCL/ (the initial state of the ATN parser) arcs 1 and 2 are attempted, and both fail. The condition on arc 1 requires that the contents of the register RELF be 1, which is the case only when a relative clause is being parsed. Similarly, the condition on arc 2 is satisfied just when a reduced relative clause is being parsed. Arc 3 succeeds, and control is pushed to the sentence net at state *S/*. 'THE AIRCRAFT WERE ENROUTE HOMEBASE' is recognized as a sentence, and an S node is returned to the declarative net where it is added to the list DCL. Control then passes to state DCL/S with 'AFTER' as the current input symbol.

4.4.2.1.2 State DCL/S. At state DCL/S, arc 1 is taken, since 'AFTER' has the feature SCONJ (sentence conjunction). 'AFTER' is added to DCL, the input string is advanced to 'CONDUCTING', and control is transferred to state DCL/CONJ.

4.4.2.1.3 State DCL/CONJ. Arc 1 at state DCL/CONJ succeeds, since 'CONDUCTING' has the feature PRESP (present participle). Control is pushed to the sentence net at state S/SUBJ, and 'CONDUCTING OPERATIONS OVER THE RED SEA' is recognized as a sentence (i.e., one with a null subject). An S node is returned to the declarative net and added to the list DCL. Control then passes to state DCL/S with the end-of-sentence marker as the current input symbol.

4.4.2.1.4 State DCL/DCL. This time arc 1 at state DCL/S fails, so arc 2, a jump to DCL/DCL, is taken. At state DCL/DCL the list DCL, which has the form given in (2), is popped.

To take another example of a slightly different form, consider (3).

(3) THE AIRCRAFT WERE ENROUTE HOMEBASE
    AFTER OPERATIONS OVER THE RED SEA.

In parsing (3), state DCL/CONJ is reached with 'OPERATIONS' as the current input symbol. This time arc 1 at DCL/CONJ fails, since 'OPERATIONS' is not a present participle, and arc 2, a push to the noun phrase net, is taken. 'OPERATIONS OVER THE RED SEA' is recognized as a noun phrase, and an NP node is returned to the declarative net and added to DCL. The list popped at state DCL/DCL will have the form given in (4).

```
(4) [S(THE AIRCRAFT WERE ENROUTE HOMEBASE),
    'AFTER',
    NP(OPERATIONS OVER THE RED SEA)]
```

4.4.2.2 The Simple Sentence. The sentence grammar (see Figure 4-4 and Appendix D) accepts sentences composed of subject, verb group, direct object, complement, and various adverbial post modifiers. It returns a node labeled 'S' which has five branches. The first branch points to the logical subject, the second to the verb group, the third to the direct object, the fourth to the complement and the fifth to a list of "verb modifiers". Given (5), for example, the sentence net returns the structure in (6).

(5) AT 0200Z 21 FEBRUARY THREE EGYPTIAN AIRCRAFT FROM RGT XB412 WERE CONDUCTING OPERATIONS OVER THE RED SEA.

(6) S(NP(THREE EGYPTIAN AIRCRAFT FROM RGT XB412),
VG(WERE CONDUCTING),
NP(OPERATIONS),
0,
[AT 0200Z, 21 FEBRUARY, OVER THE RED SEA])

(5) is parsed by the sentence net as follows:

4.4.2.2.1 State S/. From state S/, control passes via arc 1 to state PP/, the initial state of the prepositional phrase net. 'AT 0200Z' is recognized as a prepositional phrase, and a PP node is returned to the level of the sentence net, where it is added to the VMODS list. Control returns to state S/, with '21' as the current input symbol.

Back at state S/, another push to the prepositional phrase net is attempted via arc 1, but this time the push fails. Arc 2, a push to the date net, is attempted, and succeeds, with '21 FEBRUARY' being recognized as a date. An appropriate structure is returned to the sentence net, where it is placed in the VMODS list. Control passes again to state S/ with 'THREE' as the current input symbol.

Arcs 1 and 2 at state S/ are attempted in order again, and both fail. Control passes via arc 3, a jump arc, to state S/PP. The current input symbol is still 'THREE'.

4.4.2.2.2 State S/PP. At state S/PP, a push is made to state NP/, the initial state of the noun phrase net. 'THREE EGYPTIAN AIRCRAFT FROM RGT XB412' is recognized as a noun phrase, and an NP node is built and returned to the sentence net, where it is stored in the register SUBJ. Control then passes to state S/SUBJ with 'WERE' as the current input symbol.

4.4.2.2.3 State S/SUBJ. The arc at state S/SUBJ is a push to the verb group net. 'WERE CONDUCTING' is recognized as the verb group, and a VG node is returned and stored in register VGRG. Additionally, the registers PASSIVE and VHRG are raised by the post-actions PASSIVE RETR and VHRG RETR from the level of the verb group net to that of the sentence net, in order that they may be used to perform tests on the arcs at state S/VG. The value of PASSIVE is either 0 or 1 according to whether the sentence is active or passive (in the case of (1) the value of PASSIVE is 0), and VHRG contains the verb head.

4.4.2.2.4 State S/VG. From state S/SUBJ control passes to state S/VG with 'OPERATIONS' as the current input symbol. Arc 1 at S/VG is a push to the agent net, with the condition that the contents of PASSIVE be 1 and that the current input symbol be 'BY'. This clearly fails, as does arc 2, which also requires that the contents of PASSIVE be 1. Arc 3 is a push to the simple noun phrase net (state SNP/), with the condition that the contents of Passive be 0 and that the contents of VHRG have the feature TRANS (transitive). 'CONDUCTING' is transitive, so control transfers to state SNP/. 'OPERATIONS' is identified as a simple noun phrase, and an NP node is built and returned to the sentence net, where it is stored in the register OBJ. Control then passes to state S/OBJ with 'OVER' as the current input symbol.

4.4.2.2.5 State S/OBJ. The arc at state S/OBJ is a push to the verb modifier net, so control passes to state VM/. The pre-action on this arc sends the list VMODS, which already contains 'AT 0200Z' and '21 FEBRUARY', down to the level of the verb modifier net. 'OVER THE RED SEA' is recognized as a verb modifier (more specifically, a prepositional phrase) and added to VMODS, which is raised to the level of the sentence net by the post-action VMODS RETL. Control then passes to state S/S.

4.4.2.2.6 State S/S. The pop arc at S/S builds a five-branched node labeled 'S'. The first branch points to the contents of SUBJ, the second to the contents of VGRG, the third to the contents of OBJ, the fourth to the contents of COMPL, and the fourth to the VMODS list.

4.4.2.3 The Noun Phrase. The noun phrase grammar (see Figure 4-5 and Appendix D) accepts complex noun phrases with pre and post head modifiers, including adjectives, prepositional phrases and relative clauses (both full relatives, i.e. those that contain relative pronoun, and restricted relatives. It returns a four-branched node labeled 'NP'. The first branch points to a determiner phrase, the second to a list of pre-head modifiers, the third to the head noun, and the fourth to a list of post-head modifiers. Given (7), for example, the noun phrase net produces the structure given as (8).

(7) THE FOUR SR-71 RECONNAISSANCE AIRCRAFT FROM REGIMENT
XB412 WHICH DEPARTED HOMEBASE AT 0200Z WERE ENROUTE...

```
(8) NP(DP(THE FOUR),
      [SR-71, RECONNAISSANCE],
      N(AIRCRAFT),
      [FROM REGIMENT XB412, WHICH DEPARTED HOMEBASE AT 0200Z])
```

(7) is parsed by the noun phrase net as follows:

4.4.2.3.1 State NP/. At state NP/ control is pushed to state SNP/, the initial state of the simple noun phrase net. In our terminology, a simple noun phrase is one which has no post-head modifiers, i.e., one which consists of at most a determiner phrase, a list of pre-head modifiers, and a head noun. 'THE FOUR' is recognized by the simple noun phrase net as a determiner phrase, 'SR-71' and 'RECONNAISSANCE' as pre-head modifiers, and 'AIRCRAFT' as the head noun. These structures are stored in DPRG, PREMODS, and HNRG, respectively, which are raised to the level of the noun phrase net by the post-actions on the push arc. Control then transfers to state NP/SNP, with 'FROM' as the current input symbol.

4.4.2.3.2 State NP/SNP. The arc at state NP/SNP is a push to the post-head modifier net. 'FROM REGIMENT XB412' and 'WHICH DEPARTED HOMEBASE AT 0200Z' are recognized as post-head modifiers and added to the list POSTMODS, which is raised by the post action 'POSTMODS RETL' to the level of the noun phrase net. Control then passes to state NP/NP with 'WERE' as the current input symbol.

4.4.2.3.3 State NP/NP. The pop arc at state NP/NP builds a four-branched node labeled 'NP' The first branch points to the contents of DPRG, the second to the list PREMODS, the third to the contents of HNRG, and the fourth to the list POSTMODS.

4.4.2.4 The Verb Group Net. The verb group grammar (see Figure 4-6 and Appendix D) accepts the main verb, its auxiliaries and any associated evaluative adverbs. It returns a node labeled 'VG' which has four branches. The first branch points to a list of evaluative adverbs (eg. 'POSSIBLY', or 'PROBABLY'), the second to the verbal auxiliary (also a list), the third to a copula, and the fourth to the verb head. Given (9), the verb group net returns the structure in (10).

(9) ...HAVE POSSIBLY BEEN CONDUCTING FLIGHTS OVER THE RED SEA.

```
(10) VG(['POSSIBLY'],
     ['HAVE','BEEN'],
     0,
     'CONDUCTING')
```

(9) is parsed by the verb group net as follows:

4.4.2.4.1 State VG/. The arc at state VG/ is a push to the auxiliary net. 'HAVE' and 'BEEN' are recognized as auxiliary elements and placed in the list AUX, and the evaluative adverb 'POSSIBLY' is placed in the list ADVBLST. AUX and ADVBLST are raised to the level of the verb group net by the post-actions AUX RETL and ADVBLST RETL, and control passes to state VG/AUX with 'CONDUCTING' as the current input symbol.

4.4.2.4.2 State VG/AUX. Arcs 1 and 2 test for the features COPULA and BE, respectively, and both fail, since 'CONDUCTING' has neither of these. Arc 3 succeeds, and 'CONDUCTING' is stored in the register VHRG. Control then passes to state VG/VH.

4.4.2.4.3 State VG/VH. The pop arc at VG/VH builds a four-branched node labeled 'VG'. The first branch points to the list ADVBLST, the second to AUX, the third to the contents of CPRG (which in this case is empty), and the fourth to the contents of VHRG.

For (11), the verb group net returns the structure given in (12), and the parsing proceeds as follows.

(11) ...HAVE BEEN ACTIVE OVER THE RED SEA.

(12) VG([],['HAVE'],'BEEN','ACTIVE')

4.4.2.4.4 State VG/. This time the push to the auxiliary net returns an AUX list with 'HAVE' as its only member. 'BEEN' is determined not to be an auxiliary element, since it is not followed by a progressive verb form. Control passes to state VG/AUX with 'BEEN' as the current input symbol.

4.4.2.4.5 State VG/AUX. Arc 1 at state VG/AUX tests for the feature COPULA. 'BEEN' has this feature, so the transition is made to state VG/COP and 'BEEN' is placed in the register CPRG. The current input symbol is now 'ACTIVE'.

4.4.2.4.6 State VG/COP. Arcs 1 and 2 at VG/COP test for adverbs, so both of these fail. Arc 3 succeeds, so 'ACTIVE' is placed in the register VHRG and control passes to state VG/VH.

4.4.2.4.7 State VG/VH. The pop arc at state VG/VH builds the structure given in (12).

As a final example, consider the verb group in (13).

(13) FLIGHTS HAVE BEEN CONDUCTED BY AIRCRAFT FROM RGT XB412.

4.4.2.4.8 State VG/. The push to the auxiliary net retuns an AUX list containing 'HAVE'. Control passes to state VG/AUX with 'BEEN' as the current input symbol.

4.4.2.4.9 State VG/AUX. Since 'BEEN' has the feature COPULA (although in (13) it is not used as such), arc 1 succeeds and 'BEEN' is stored in CPRG. Control passes to state VG/COP with 'CONDUCTED' as the current input symbol.

4.4.2.4.10 State VG/COP. Arcs 1, 2 and 3 at state VG/COP each fail, since 'CONDUCTED' is neither an adverb nor an adjective. The parser backs up to state VG/AUX, undoes the action on arc 1, and attempts arc 2 with 'BEEN' as the current input symbol. Arc 2 succeeds, so the transition is made to state VG/BE with 'CONDUCTED' as the current input symbol.

4.4.2.4.11 State VG/BE. Arcs 1 and 2 at VG/BE both fail. Arc 3, which tests for the feature PASTP (past participle), succeeds, so 'CONDUCTED' is stored in VHRG and PASSIVE is given the value 1. Control passes to state VG/VH with 'BY' as the current input symbol.

4.4.2.4.12 State VG/VH. The pop arc at state VG/VH builds the structure given in (14). In addition, the contents of PASSIVE will be used at the level of the sentence net to determine that (13) is a passive, and the sentence net will thus return the structure given in (15).

```
(14) VG([],['HAVE'],0,'CONDUCTED')

(15) S(NP(AIRCRAFT FROM RGT XB412),
     VG(HAVE CONDUCTED),
     NP(FLIGHTS),
     0,
     [])
```

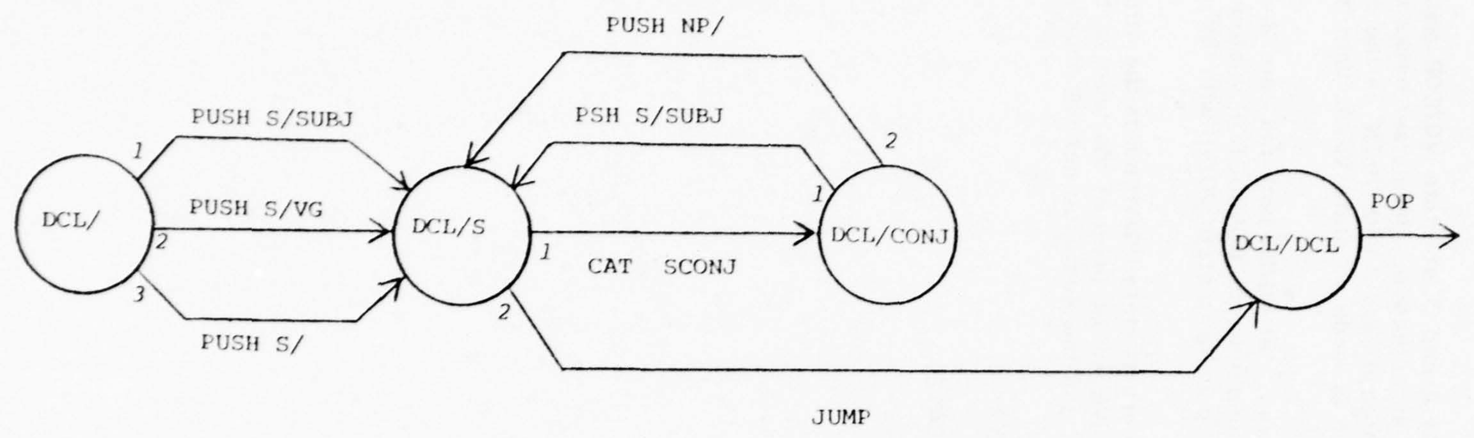


Figure 4-3. The Declarative Net

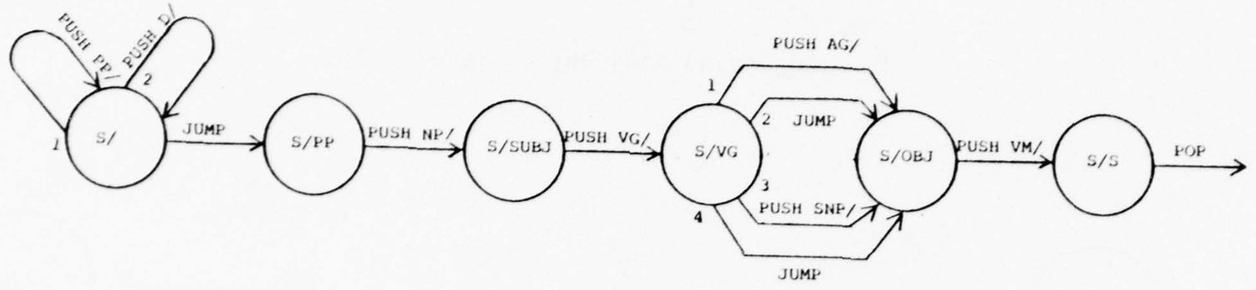


Figure 4-4. The Sentence Net

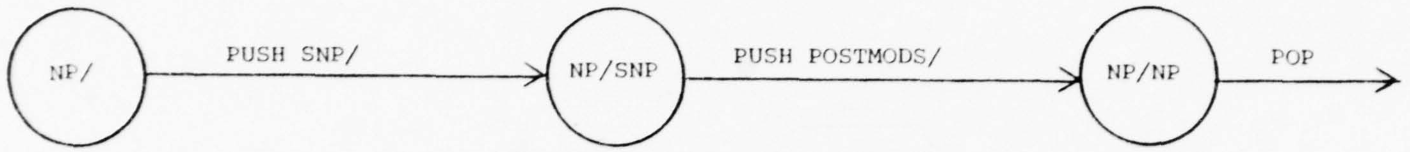


Figure 4-5. The Noun Phrase Net

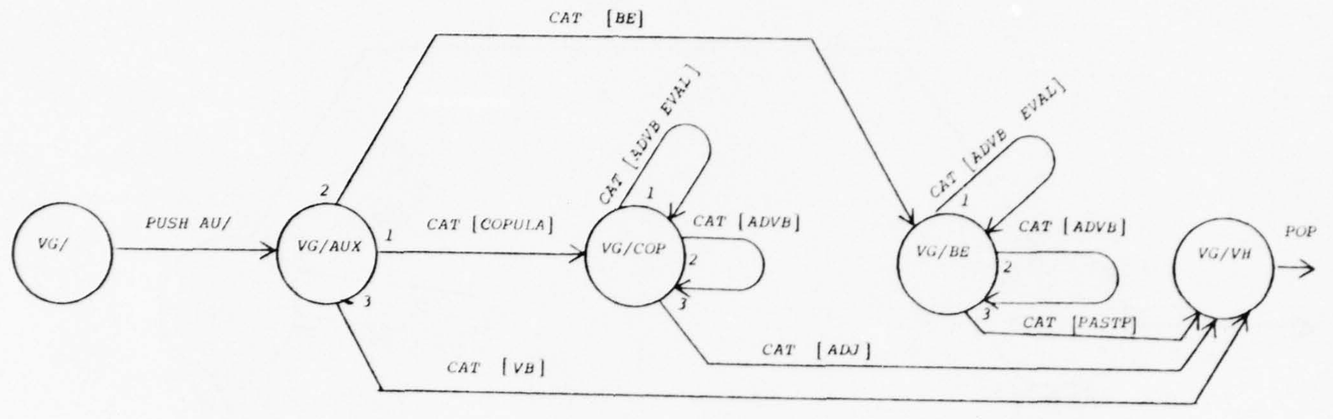


Figure 4-6. The Verb Group Net

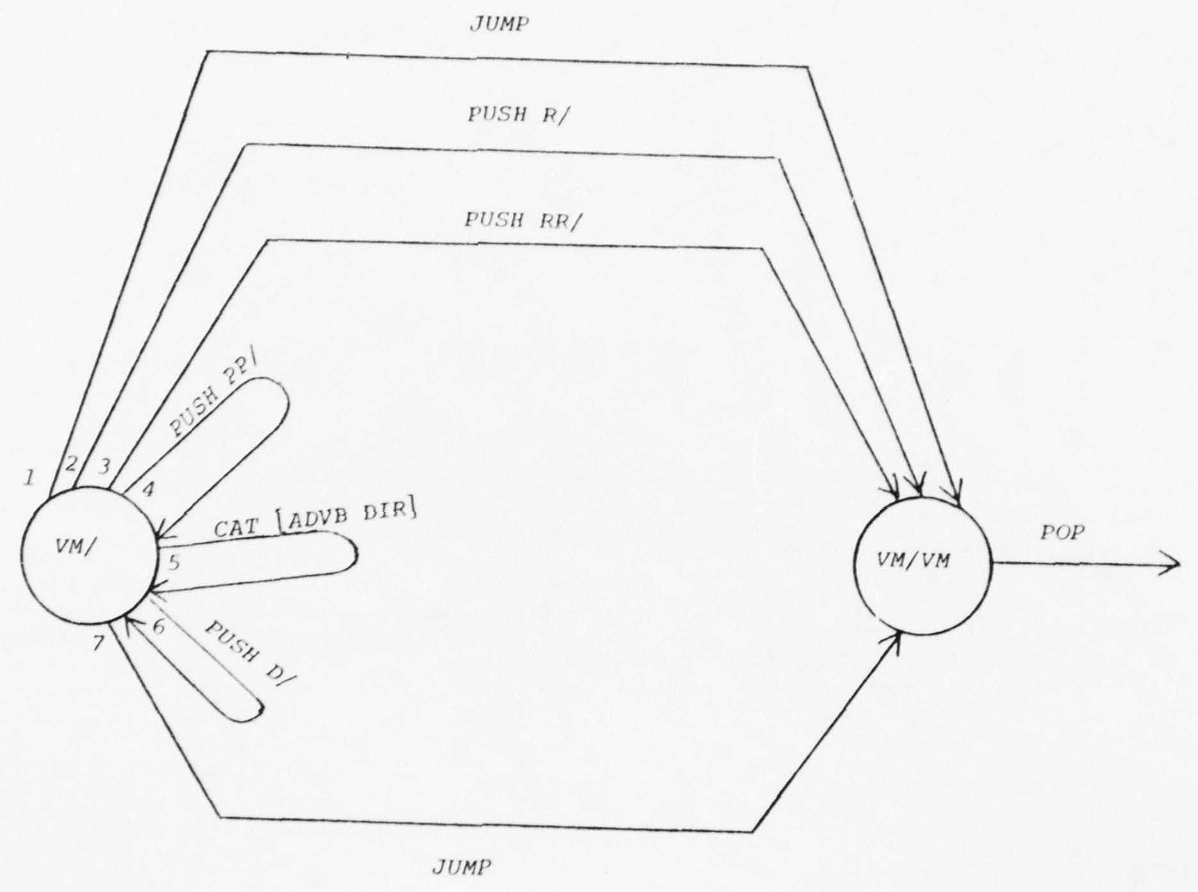


Figure 4-7. The Verb Modifier Net

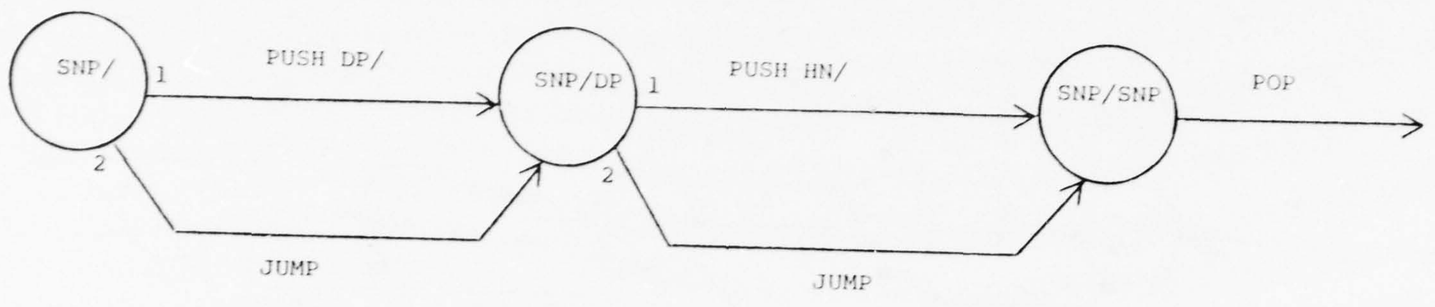


Figure 4-8. The Simple Noun Phrase Net

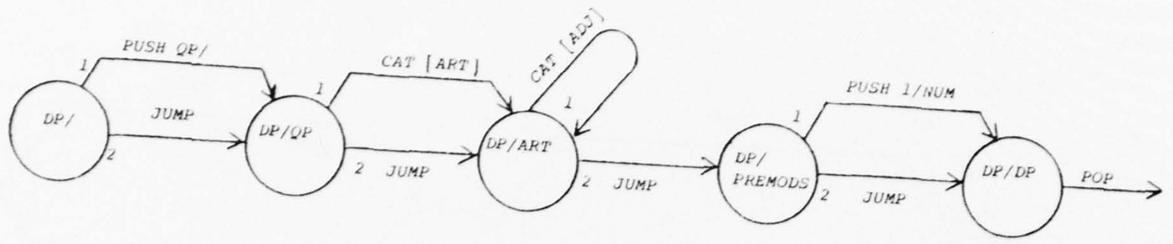


Figure 4-9. The Determiner Phrase Net

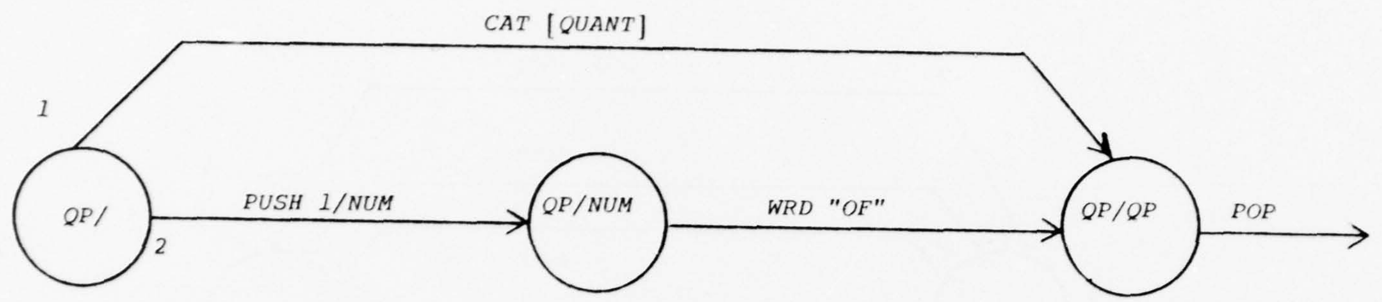


Figure 4-10. The Quantifier Phrase Net

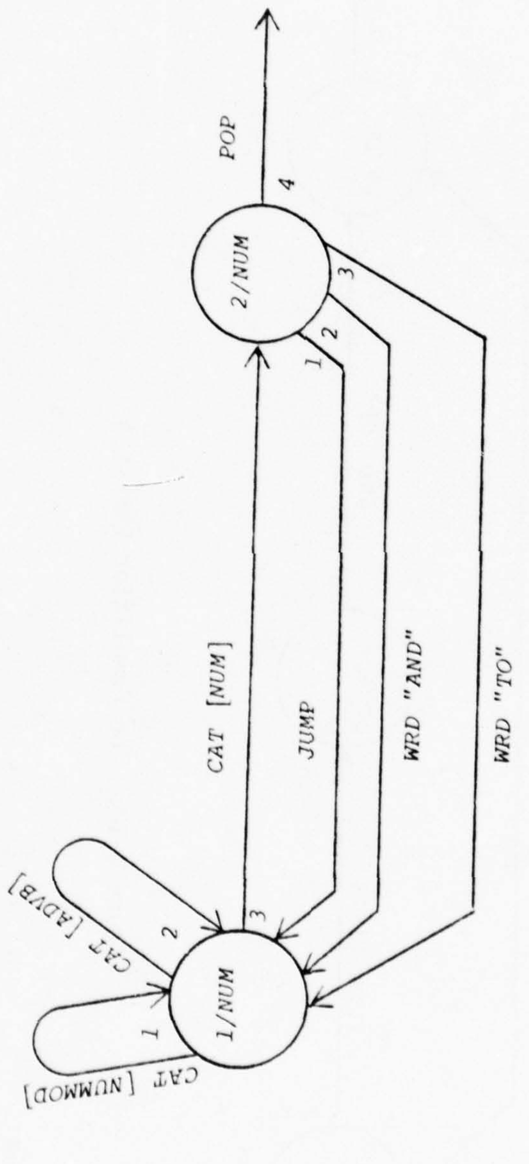


Figure 4-11. The Number Net

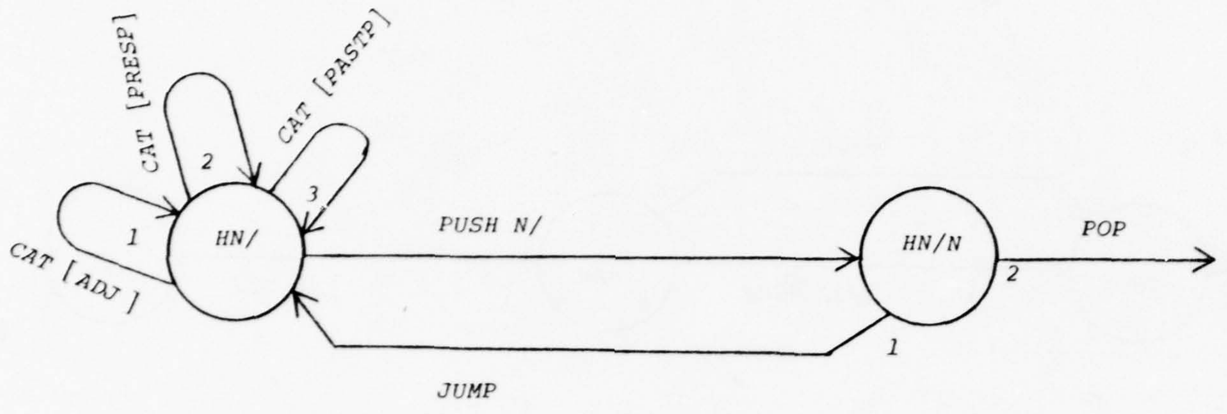


Figure 4-12. The Head Noun Net

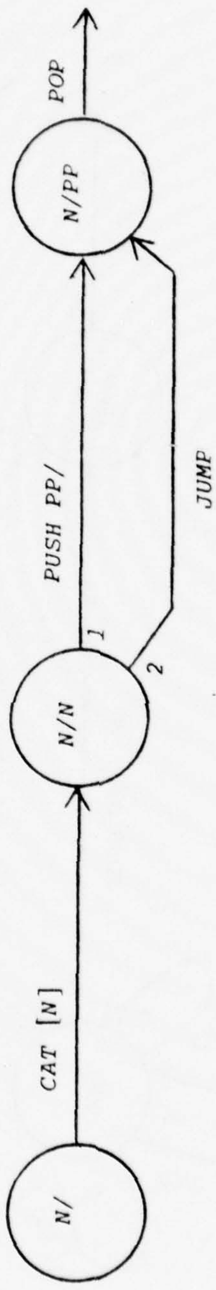


Figure 4-13. The Noun Net

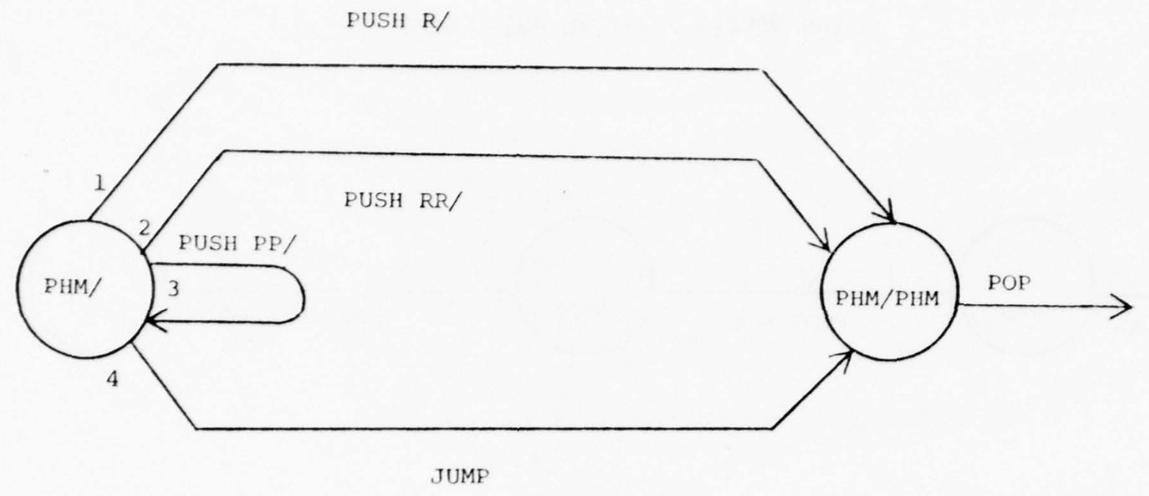


Figure 4-14. The Post-Head Modifier Net

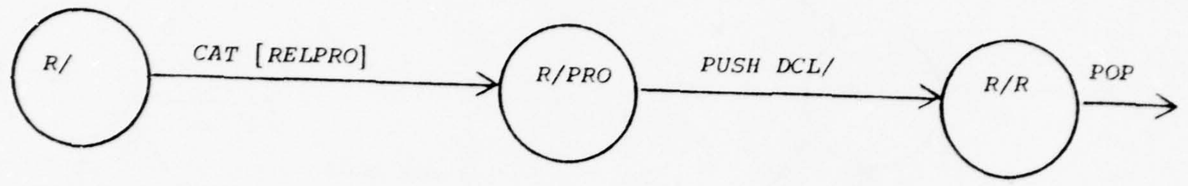


Figure 4-15. The Relative Clause Net

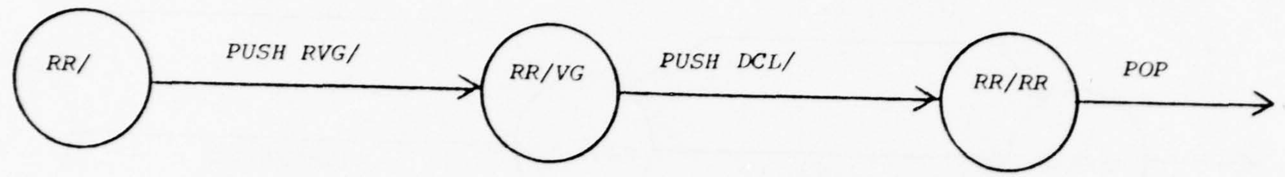


Figure 4-16. The Reduced Relative Clause Net

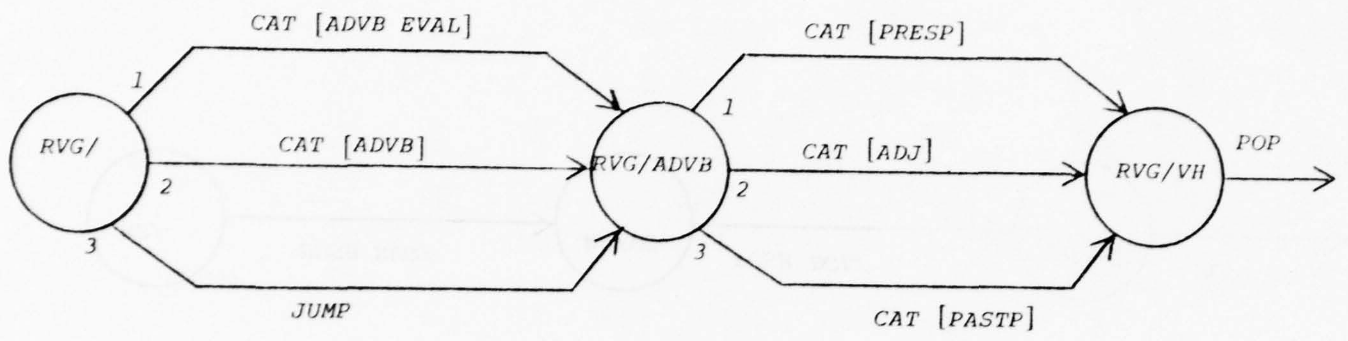


Figure 4-17. The Verb Group Net for Reduced Relative Clauses

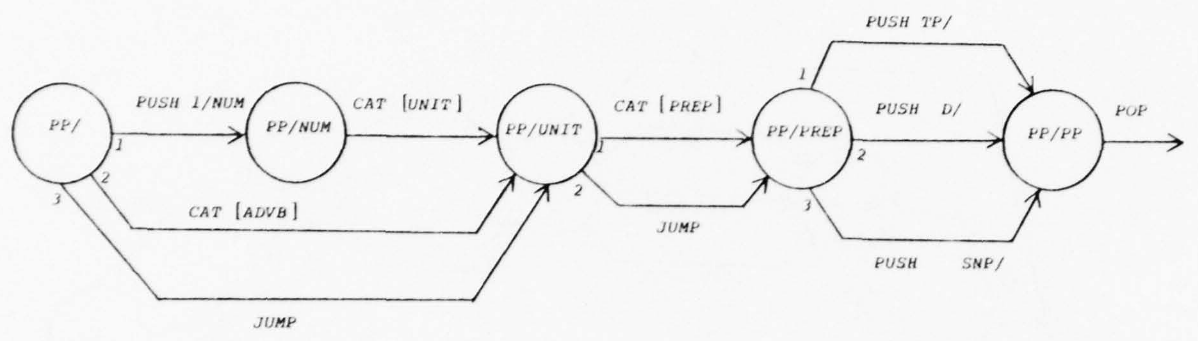


Figure 4-18. The Prepositional Phrase Net

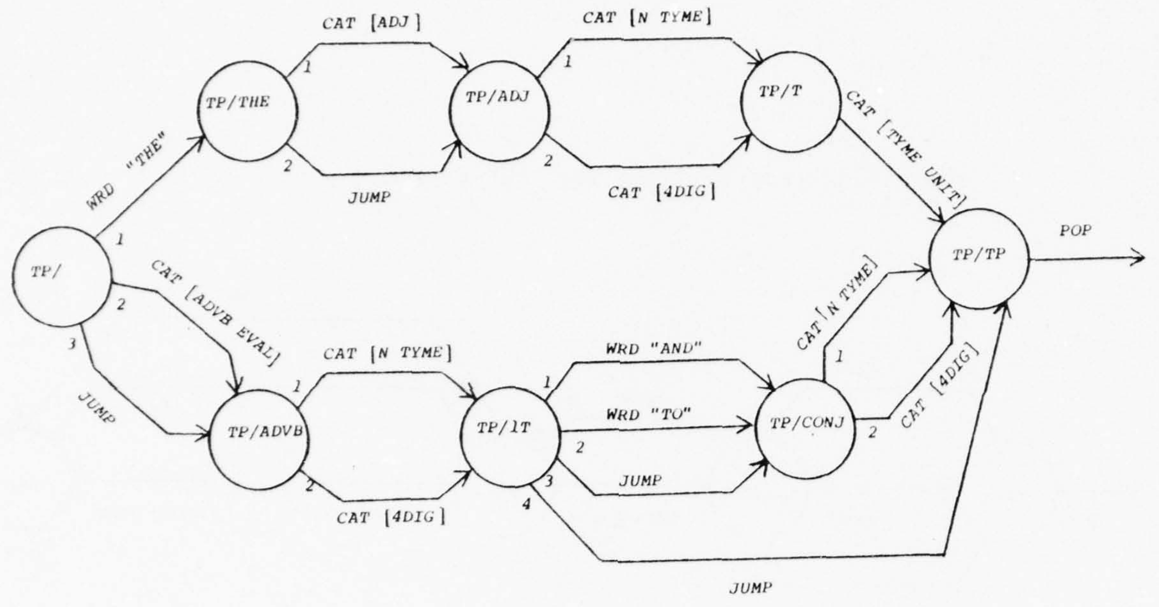


Figure 4-19. The Time Phrase Net

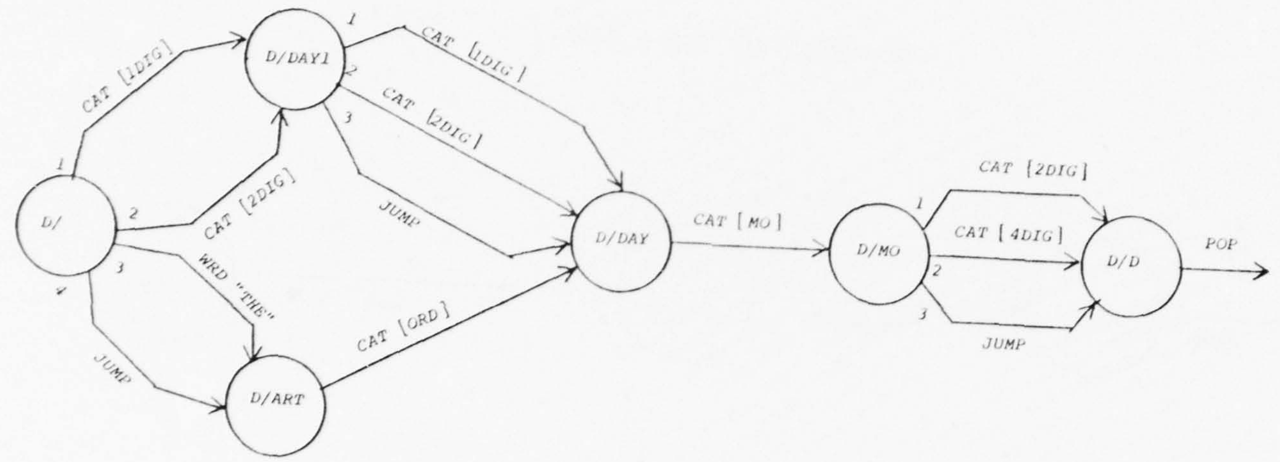


Figure 4-20. The Data Net

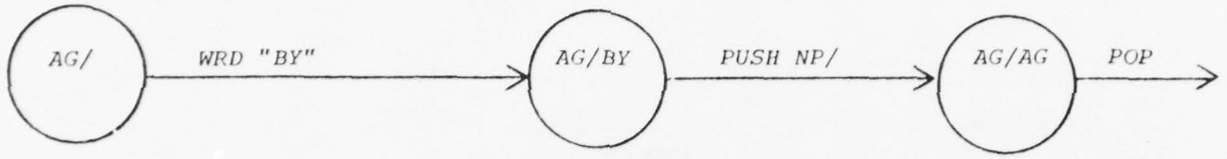


Figure 4-21. The Agent Net

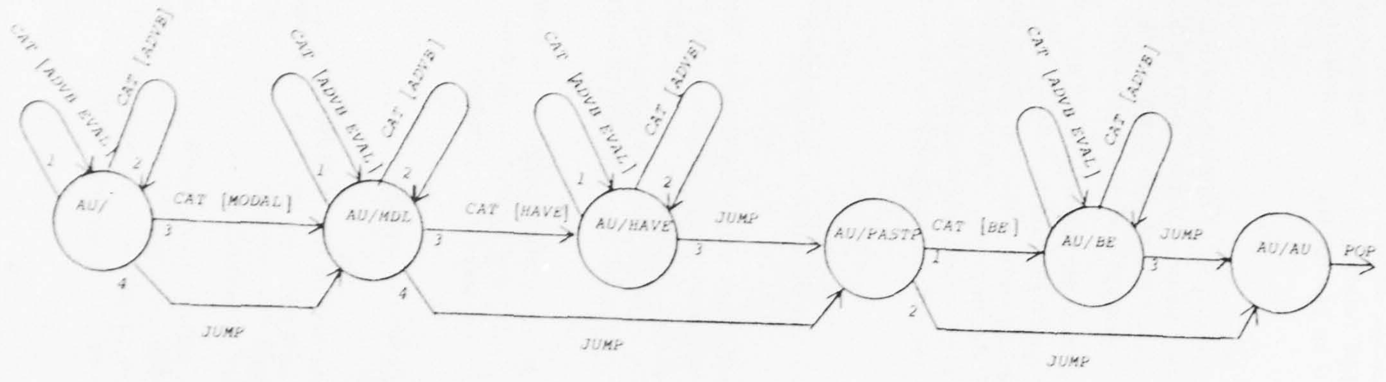


Figure 4-22. The Auxiliary Net

4.5 Template Descriptor Selection: Methodological Issues

This section provides a general discussion of some fundamental issues pertaining to the selection of descriptors for templates relating to any subject domain in general, and lists the descriptor system developed so far for the domain of aircraft activities in particular.

4.5.1 User-Related Considerations

The set of properties used for the description of events relating to a particular subject domain must answer the what, who, where, when, and why information questions relevant to the analyst's task. The definition of the descriptors and their organization, therefore, must be consonant with the analyst's view of the world.

In general, any number of properties may be specified for any given class of entities. However, not all properties have the same degree of usefulness in a given context. The properties selected for inclusion in a template must, therefore, be sensitive to the task the template is designed to support. Accordingly, the first criterion for selection is that of relevance. Templates must include only that information which is particularly relevant and useful to the task at hand, and not the full range of facts one might find in an encyclopedia.

4.5.2 Linguistic Considerations.

In this subsection the discussion will evolve around the linguistic criteria for descriptor selection.

Broadly speaking, descriptors fall into two major categories those that involve "deep case" relations, and those that involve inferences of a special kind. "Deep cases" are binary relations which specify an event regardless of the surface realization of that event description as a sentence or a noun phrase. The descriptors involving inferences are restricted to those which have to do with the relations of entailment and presupposition.

Descriptors selected for inclusion in templates within a particular subject domain are pragmatically determined from a linguistic and logical analysis of a representative sample of intelligence messages. The criterion used for selection of "deep case" relations is the following:

> A deep case is a relation whose value is usually
> specified for a given event type.

Thus, flight reports include a description of the object(s) which is (are) doing the flying and frequen ly mention other relations such as the source of the flight, its direction, the area overflown, the destination, and the mission. These properties are assigned the status of "deep cases" in the sense specified above.

Pilots, however, or navigators, are very seldom mentioned in flight reports. They will be treated differently, namely, they will be regarded as presupposition of the flight event. The notions of entailment and presupposition are explicated in a later subsection. The next section discusses the notion of "deep cases", which is the basis for defining intra-template relations.

4.5.2.1 The 'Deep Case' System.

A "deep case" is a binary relation which holds between a predicate (usually, but not necessarily, realized as a verb) and one of its arguments. Deep cases are used both in accounting for the relative acceptability of natural language sentences and in explaining how an intelligent system might understand language. This is done in terms of a "case structure" and "selectional restrictions". The case structure for any predicate is the set of cases allowed in a description of that predicate. Selectional restrictions then place semantic constraints on the objects which fill the case slots.

Each predicate has a number of cases. These may include adverbial modifiers, temporal indicators, and other propositions as well as the usual nominal cases. For example, the case structure for the predicate "be enroute" might be (Object, Destination), where each case may appear at most once. Object represents the notion "The thing which is enroute". The meaning of Destination is clear. Both the Object and the Destination must be present in the message text; i.e., they are obligatory cases which are required for the event description to make sense.

Other predicates may have allowable cases which need not necessarily be realized in the text. Such a predicate is "fly", for which only the "thing which is doing the flying" is obligatory. The other allowed deep cases, such as Source, Destination, Extent, Direction, Area, Mission, etc., are optional, i.e., they may or may not appear in the actual text. Any of the following sentences satisfies the descriptor structure for the fly template.

> The aircraft flew south.
> The aircraft flew to Mombasa.
> The aircraft flew from London to Cairo.
> The aircraft flew as far south as Cairo.
> The aircraft flew a reconnaissance mission over Uganda between
> 0012 and 0036 on 26 Feb 1975.

However, if the first and last sentences refer to a single aircraft, based on one message or more than one, the additional information provides material to complete the empty descriptor slots in the 'flight' template representing that event. Selectional restrictions vary from global constraints on the use of a case (e.g., "every agent must be animate") to local constraints on the use of a case with a particular predicate (e.g., "the destination of a flight must be a geographic location such as a country, a city, or an airport").

The degree to which a case-based theory can account for the correct interpretation of text depends upon the way the cases mediate between surface forms and conceptual structures. The transformation of surface forms into meaning representations is the function of the procedural component of templates, which was described in Section 3.

4.5.2.2 Presupposition and Entailment.

Presupposition and entailment are a subclass of inferences which appear to be closely connected with the structure of language. They arise from two main structural sources: one, the semantics of particular words, and two,from the syntactic (or relational) structure of sentences.

4.5.2.2.1 Entailment. A proposition P entails a proposition P' if and only if in every context in which P is true, P' is also true. For example, a plane cannot fly unless it has taken off, cannot land unless it has been flying, must be in flight if it has taken off and has not landed or been destroyed. Thus a take-off event entails a subsequent flight, while a flying event entails a preceding take-off. A landing event entails a preceding flight, while a flying event entails a subsequent landing.

The above entailment relations are obligatory and specific to the respective event predicate, i.e., a flight entails a previous take-off because of the meaning of "fly", while a landing entails a previous flight because of the meaning of "land".

Such entailments predict the normal, expected, ordering of events in the air activities world. Any violation of these expectations can serve as a warning to the analyst that some external force has altered the predicted course of events.

For example, if a plane which is reported in flight does not land within expected limits of time, it may have altered course, may have made a forcedlanding, or may have been destroyed. It is important that the analyst be alerted to any deviation from the expected.

4.5.2.2.2 Presupposition. A second, related concept is the notion of presupposition. A proposition P (logically) presupposes a proposition P' if and only if P entails a P' and ~P entails P'. Therefore, whether P is true or false, P' must be true if P is to make any sense at all. It is clear from the above definition that all logical presuppositions P' are also entailments of P. Presuppositions play an important part in the meaning of many words.

For example, in the air activities domain, a flying event presupposes that the thing which does the flying is an aircraft. The presupposition is related to selectional restrictions and is incorporated in the specification of what may fill the Object slot of the FLY event.

Certain aspectuals (e.g., begin, continue, end) are also associated with presuppositions. For example, both the sentence "the plane continued flying" and its negation"the plane did not continue flying presuppose that at some point the plane was flying.

The predicate "return" presupposes that the object which is reported to have returned has been at that location before.

One of the important aspects of presupposition in language is that it informs the reader that the presupposition must be considered true. Thus, if some aircraft is reported to have returned to its normal operating area, it must be considered true that some time before its return it took off from that particular area. Even if the report were negative, i.e., stating that the aircraft in question had not returned to its normal operating area, the presupposition that it had previously taken off from that area remains true.

Thus, presuppositions and entailments add information which is conceptually associated with some entity, but is very seldom mentioned explicitly.

This fact can be of assistance to the analyst in establishing the identity of objects involved in events reported by different sources in different ways, or perhaps in seeking to establish links between events which otherwise might appear unconnected.

The descriptor system for the air activities sublanguage then, includes, in addition to those discussed previously, the two descriptors related to inferences, namely, entailments and presuppositions.

4.5.3 The Descriptor System for the Aircraft Domain Table 4-24 shows the descriptor system so far developed for the air activities sublanguage.

Table 4-24. Air Activities Descriptor System

A. Motion related descriptors	
Agent	Animate instigator of the action.
Object	The entity that moves or changes or whose position or existence is being described.
Source	The location of the object at the beginning of a motion.
Destination	Projected or actual destination of the object at the end of the motion.
Direction	Direction of motion of object at time of observation.
Path	Path or area traversed during motion.
Extent	Extent of motion.
Limit	Limit of motion.
Altitude	Altitude of object at time of observation.
Region	General location of the action.
Status	Begin, continue, end.
Time specification	Time of observation or duration of the event.
B. Event related descriptors	
Mission	Purpose of flight.
C. Aircraft related descriptors:	
Equipment	
Class	
NATO designation	
Nationality	
Subordination	
Homebase	
Staging base	
Set specification	
Configuration	
D. Inferences	
Entailments	
Presupposition	
The latter include objects normally associated with some concepts but very seldom mentioned, (e.g., pilot, fuselage).	

5.0 References

Colmerauer, A. *Les Grammaires de Metamorphose.* Groupe d'Intelligence Artificielle, Marseille, Marseille-Luminy, Nov. 1975.

Dahl,V. *Un Systeme Deductif d'Interrogation de Banques de Donnees en Espagnol.* Groupe d'Intelligence Artificielle, Universite de Marseilles-Luminy, Nov,1977.

Gimpel, J. *Algorithms in SNOBOL4*, Wiley & Sons, N.Y., 1976.

Kowalski, R.A. *Logic for Problem Solving.* DCL Memo 75, Dept of AI, Edinburgh, March, 1974.

Kuhns, J.L. and C.A. Montgomery, *Synthesis of Inference Techniques, Event Record Specification System Concept: Preliminary Notions*, OSI Technical Report No. 1, R73-008 (Contract No. F30602-73-C-0333), Operating Systems, Inc., 17 October 1973.

Kuhns, J.L., *Synthesis of Inference Techniques: An Interpreted Syntax for the Logical Description of Events*, OSI Technical Note No. 2, N74-003 (Contract No. F30602-73-C-0333), Operating Systems, Inc., 31 May 1974.

Kuhns, J.L., C.A. Montgomery and D.K. Welchel, *ERGO -- A System for Event Record Generation and Organization*, RADC-TR-75-51, March 1975.

Morrison, Donald R. *PATRICIA - Practical Algorithm to Retrieve Information in Alphanumeric*, Journal of the ACM, Vol. 15, No. 4, 1968.

Pereira, L.M., F.C.N. Pereira and David H.D. Warren. *User's Guide to DECsystem-10 Prolog.* Provisional Version, April, 1978.

Reichenbach, H. *Elements of Symbolic Logic*, Macmillan Co., N.Y., 1947.

Robinson, J.A. *A Machine-Oriented Logic Based on the Resolution Principle.* Journal of the ACM, 12, 1965.

Roussel, P. Prolog: *Manuel de Reference et d'Utilisation.* Groupe d'Intelligence Artificielle, Marseille-Luminy, 1975.

Russell, B. *On Order in Time*, reprinted in *Logic and Knowlege* (R. C. Marsh, ed.), George Allen and Unwin, London, 1956.

Silva, G. and C.A. Montgomery, *Automated I&W File Generation*, RADC-TR-77-194, June 1977.

Silva, G. and C.A. Montgomery, Knowledge Representation for Automated Understanding of Natural Language Discourse. Computers and the Humanities, Vol 11, Pergamon Press, 1978.

van Emden, M.H. *Programming with Resolution Logic.* Report CS-75-30, Dept. of Computer Science, University of Waterloo, Canada, Nov, 1975.

Warren D. H. D. *Implementing Prolog- Compiling Predicate Logic Programs.* Dept. of AI Research Reports 39 & 40, Edinburgh, May, 1977(a).

Warren D.H.D. *Logic Programming and Compiler Writing.* Dept. of AI Research Report 44, Edinburgh, September, 1977(b).

Warren, D.H.D., Pereira, L.M. and Pereira, F.C.N. *Prolog- The Language and its Implementation Compared with Lisp.* Procs. of the ACM Symposium on AI and Programming Languages, SIGPLAN/SIGART Newsletter, Rochester NY, Aug 1977.

Weiner, N. *A Contribution to the Theory of Relative Position*, Proc. Camb. Phil. Soc., vol 17,1914.

Woods, W.A. *Procedural Semantics for a Question Answering Machine.* AFIPS Conference Proceedings, Vol. 33, 1968.

Woods, W.A. *Progress in Natural Language Understanding: An Application to Lunar Geology.* AFIPS Conference Proceedings, Vol. 42, National Computer Conference, 1973.

Woods, W.A. *Transition Network Grammars for Natural Languge Analysis.* Communications of the ACM, October, 1970.

Woods, W.A. *What's in a Link: Foundations for Semantic Networks.* in Daniel G. Bobrow and Allan Collins (eds) *Representation and Understanding*, New York, Academic Press, 1975.

1.0 Introduction

This section describes the implementation of the concepts of the first section, in the form of the MATRES II system. The following subsections describe the various components of the system. In some places, references are made to MATRES I, the product of the contract directly preceding the current one. The reader is referred to the final report of that contract for details.

Subsection 2 presents the data structures and algorithms for the sentence input and grammar processing vocabulary; this vocabulary is essentially an extensive modification of the MATRES I system. Subsection 3 describes the capabilities added in the area of morphology. The implementation of the ERL evaluation process, including the abstract machine which is the target language of ERL, is described in Subsection 4. The ERL compiler, which is the only non-Forth module, is discussed in Subsection 5. The last three subsections are intended as a guide to the Forth program files listed in Appendix A, and provide glossaries of the Words in those files.

2.0 Design of Lexical and ATN Processors

2.1 General Principles

All operations involving addresses use a set of Words which expect addresses, even if the operations are simple; e.g. to increment an address "p" by "n" words (bytes), use "p n a+" ("p n \a+"), where "a+" and "\a+" are defined as:

```
: a+ 2* + ;   : \a+ + ;
```

Note that these words are not commutative, and expect the address on 2OS.

Of course, any FORTH Word may be used as an action, but it should be remembered that actions may have to be undone; therefore, action Words should not modify storage outside of the context blocks.

For the purposes of defining structure, we assume that structures will only be built and examined, not modified or deleted; all dynamic storage will be released when a sentence has been completely processed. Later enhancements may require a dynamic space reclamation mechanism, but we don't make any provision for that now.

2.2 Data Structures

All the following structures reside in block storage to provide for uniformity of addressing, since we use a 16-bit address for block storage similar to the one used in I&W II for data base pointers; these pointers cannot be distinguished from core addresses by their content alone.

The lexical unit, as defined below, is different from that of I&W II; here we treat different senses of words having multiple senses as distinct lexical units, and use forward, backward and alternate pointers to link units within the sentence, thus creating a "two-dimensional" list of lexical units for the sentence.

A *lexical unit* has a string pointer, a string length, a forward pointer (to the next unit in the sentence), a backward pointer (to the previous unit), an alternate pointer (to the next sense unit for the current word or phrase), and a sense.

A *register* is empty or contains a pointer to a lexical unit or a list or a node.

A *node* has a label and one or more branches; a *branch* is empty or points to a lexical unit or a list or a node; a *label* is the name of a Word which contains a branch count and the label name (in the format of an ERL functor literal; see the section on ERL).

A *list* has a zero branch count and a link to the first listel; a *listel* has a pointer to a lexical unit or a list or a node, and a link; a *link* points to a listel or is zero.

2.3 Action Definitions

The actions for I&W III are totally different from the previous ones; for convenience, we will repeat the syntax definitions from MATRES here, with new rules for declarations and actions (note that "␢" represents a blank that must be present).

grammar ::= 'GRAMMAR␢' start-state-name declaration* state+ '␢ENDGRAMMAR'

start-state-name ::= state-name

declaration ::= '␢REGISTER␢' register-name | '␢LIST␢' list-name | number '␢LABEL␢' label-name

```
register-name, list-name, label-name ::= Word

state ::= ':Sb̸' state-name arc+ 'b̸;;'

state-name ::= Word

arc ::= ':WRDb̸' '"b̸' string '"' tail |

        ':MEMb̸' '(b̸' ( '"b̸' string '"' )+ 'b̸)' tail |

        ':CATb̸' [ '-' ] '[b̸' feature+ 'b̸]' tail |

        ':TSTb̸' condition 'b̸!!b̸' action* 'b̸=>b̸' state-name 'b̸,,b̸' |

        ':PSHb̸' [ condition 'b̸!!b̸' ] action* 'b̸TOb̸' state-name action* 'b̸=>b̸' state-name
          'b̸,,b̸' |

        ':POPb̸' [ condition 'b̸!!b̸' ] action* ptr 'b̸,,b̸' |

        ':JUMPb̸' state-name [ condition 'b̸!!b̸' ] action* [ 'b̸ADVb̸' | 'b̸RETb̸' ] 'b̸,,b̸'

tail ::= [ condition ] 'b̸!!b̸' action* 'b̸=>b̸' state-name 'b̸,,b̸'

condition ::= condition condition ( 'b̸ANDb̸' | 'b̸ORb̸' ) | condition 'b̸NOTb̸' | cond

cond ::= pos test | Word

test ::= [ '-' ] '"b̸' string '"' | [ '-' ] '[b̸' feature+ 'b̸]' | 'b̸[EOS]b̸'

action ::= Word | ptr register-name 'b̸SETRb̸' | register-name 'b̸GETRb̸' | ptr list-name
        'b̸ADDLISTb̸' | ptr register-name 'b̸SENDRb̸' | register-name 'b̸RETRb̸' | list-name (
        'b̸SENDLb̸' | 'b̸RETLb̸' )

ptr ::= pos | register-name 'b̸GETRb̸' | list-name | ptr+ label-name 'b̸NODEb̸'

pos ::= 'b̸*b̸' ! 'b̸*+1b̸' ! 'b̸*-1b̸' | register-name 'b̸GETRb̸'
```

A "ptr" construction returns to the stack a pointer to a lexical unit, a list, or a node; in the latter case, the node is actually created by the Word NODE from the label and "ptr"s on the stack. The label preceding NODE must have been declared by a LABEL declaration which gives the number of pointers to take from the stack; for example, the declaration "4 LABEL THING", together with the action "REG @ *+1 *-1 LIST THING NODE" creates a node labelled THING with four branches, the first pointing to the list LIST, the second to the previous lexical unit, the third to the next lexical unit, and the fourth to what REG points to. A pointer to the list is returned to the stack. Note that the order of pointers in the node is the reverse of the order in which they appear in the text.

ADDLIST adds a ptr to the front of the specified list; thus, as with nodes, the elements of the list will be in reverse order from that which they were added.

SETR is used to set a register to a value, and GETR is used to retrieve the value of a register. GETR may be used in tests, with registers which point to lexical units.

SENDR and SENDL are used only in the preactions of a PUSH node. SENDR sends a value to a register at the level of the subnet (registers and lists are normally empty on entry to a subnet). Similarly, SENDL sends the current value of a list to the subnet level.

RETR and RETL are used only in the postactions of a PUSH node, and are complementary to SENDR and SENDL, in that they retrieve register and list values, respectively, from the subnet values at the time of the POP.

A POP arc must have a "ptr" as its last (or only) action, which will cause the "ptr" to be assigned to * at the next level up.

2.4 Internal Structure and Algorithm Specifications

2.4.1 Layout of Block Storage All the system structures except the compiled ATN reside in block storage to provide for uniformity of addressing, since we use a 16-bit address for block storage similar to the one used in I&W II for data base pointers; these pointers cannot be distinguished from core addresses by their content alone.

The first structure in block storage is the lexicon, starting at the block specified by the constant SLEX. The variable ELEX holds a pointer to the last byte of the lexicon. Next, starting on the next block boundary, will be the text of each input sentence, followed by the chart for that sentence, and then the stack of state frames. The base block number for all pointers except within the lexicon will be contained in the constant SBASE.

2.4.2 Structure of the Input Sentence As described below, each input sentence will be read and stored in FORTH block storage in two parts: the actual character string comprising the sentence, and a structure of entries corresponding to the lexical units (words or phrases) found in the sentence. Each such entry has the following structure:

Item	Length
Address of character string for unit	1 word
Length of character string (in bytes)	1 word
Pointer to next unit	1 word
Pointer to previous unit	1 word
Pointer to alternate unit	1 word
Feature vector	NWRD words

At the end of the list of entries is a "pseudo-entry" consisting of all zero entries except for the previous unit pointer, to mark the end of the sentence.

This structure allows for a list of alternate senses for a given word in the sentence, and also for handling phrases. For example, it may or may not be appropriate to treat a given sequence of words as a single lexical unit at a particular place in a sentence; with this structure, we could build, as alternates, both the lexical unit corresponding to the phrase interpretation and the list of units corresponding to the string of words (although we don't do that in this system). We will call the structure built for a sentence the *chart* of the sentence.

The following variables are set to provide access to these structures:

TXTP points to the first character of the sentence text.

SENTP points to the first lexical unit of the sentence chart, and also marks the end of the sentence text.

FRAMSTRT points to the base of the first state frame for the sentence, and also marks the end of the chart for the sentence.

2.4.3 Text Input and Sentence Construction Text input is performed by the Word GETTXT and its auxiliary Words. The FORTH Word WORD is used to get the next string of nonblank characters, and CHKWRD strips off trailing commas and periods, and returns an indication if the punctuation was a period; this is used to terminate the sentence text. Words will be separated by one blank, and a period set off by blanks will be appended to the text (by TEXFIN).

MAKESENT makes the chart of the sentence, creating a lexical unit for each sense of each word or phrase found in the lexicon, and linking them as appropriate. Its operation is discussed in more detail in the next section.

2.4.4 ATN Processor - Data and Program Structures The ATN compiler produces a Word in the dictionary for each state, having code structures for each arc; these are described below. By "code" is meant a COLON-style sequence of Words implementing a particular operation. Optional elements of a structure are delimited by square brackets - "[]". All states are encompassed in a scope block named "GRMDF".

2.4.4.1 State Structure

State entry code
Arc structures for arcs of state
End-of-state code

2.4.4.2 Arc Structures (by arc type)

```
WRD, MEM,     Arc entry code
CAT, TST    Code for tests - leaves 1 or 2 truth values on stack
          Code for !!
          [Code for actions]
          Code for => and state name

PUSH        Arc entry code
          [Code for tests
          Code for !!]
          [Code for preactions]
          Code for TO and state name
          [Code for postactions]
          Code for => and state name

POP         Arc entry code
          [Code for tests
          Code for !!]
          [Code for actions]
          Code for returning to calling arc

JUMP        Arc entry code - 3rd and 4th words are skip around state
           name
          [Code for tests
          Code for !!]
          [Code for actions]
          [Code for ADV or RET]
          Code for jumping to state
```

In all arc types, the first two words of the"arc entry code" comprise a word pointing to the next arc (or to the end-of-state code), then the FORTH-style code word (a pointer to COLON).

2.4.5 Operation of the ATN Processor The operation of the ATN executive processor for state-to-state transitions would be almost trivial were it not for the fact that, for natural language processing, the ATN must be considered non-deterministic. For example, it is possible to have two arcs leaving a given state with the same tests but different destination states. As a result, it must be possible to backtrack along a path to a previous state, undoing any actions along the way.

In MATRES, this is implemented as follows: when an arc is successfully traversed, the current computation context is pushed onto the dictionary, and a new context established. Thus the stack of contexts describes the currently active path through the ATN. When the last arc in a state is tried unsuccessfully, backtracking is done by simply popping the stack and restoring the previous context, which includes a specification of the next arc to try in the (once again) current state.

The context referred to above, which will subsequently be referred to as a *state frame* to avoid confusion with the linguistic sense of "context", comprises a base pointer (FRAMBASE) and a set of value cells associated with various item names. Each name is defined as an offset from the base pointer by the ITEM defining Word. The dictionary pointer is kept pointing just past the current frame, and the base pointer is pointing to a "hidden" cell containing the previous base pointer; thus a new state frame may be defined by simply moving the base pointer to agree with the dictionary pointer and setting the dictionary pointer above the item with the highest offset, and an old frame restored by setting the dictionary pointer to the base pointer and restoring the base pointer from the cell it points to. This scheme allows the dynamic addition of data to the current frame by simply advancing the dictionary pointer.

The following data items are present in each stack frame:

STAR:	special register, kept identical to LEX except on return from a PUSH
LEX:	pointer to current lexical unit
STAAT:	pointer to current state structure
ARC:	pointer to current arc structure
ARCNO:	current arc number within state; used for trace printing
RETRN:	contains the IC to return to on POP, and the base pointer associated with the state from which the last PUSH was done

Register values and list heads are allocated above these items, followed by dynamic allocations of nodes and list elements.

The Word NEWFRAME is used to establish a new frame. It sets up a new state frame as described above and copies all the previous item data into it. The complementary word, OLDFRAME, restores the previous frame as described above.

The actual transition to a new state is done in two steps. First, the code doing the transition, which has the name of the new state stored within it, calls FNDSTATE, which finds that name in the dictionary and returns its code word on the stack and in STAAT. Next, a SKPTO is called, and it transfers control to the address on the top of the stack. Note that this is done at the same level of the return stack.

2.4.5.1 Algorithms for ATN Processor Elements

Element	**Algorithm**
State Entry	Set ARC to first arc in state, go through ARC
Arc Entry	NEWFRAME
!!	If two conditions present, AND them;
	If top of stack true, put pointer to next lexical unit on stack (current unit if JUMP arc); if PUSH arc, set SNDP to top-of-dictionary in case a SENDR is done;
	If false, OLDFRAME, set LEX to next alternate lexical unit (if no next alternate, reset to first alternate, update ARC), go through ARC
=>	Top of stack->LEX, get called state, go to it
TO	Assign "IC+1" and previous base pointer to RETRN, clear registers and lists, move in any SENDRs from top of dictionary, get called state, go to it
POP Return	Get IC and base pointer from RETRN, copy base pointer's registers and lists, arc, state, and return data into current frame, TOS into STAR, go to postactions through IC
End of State	Treat like false branch of !!
ATN Start	Set: LEX to first lexical unit, STAAT to ATN start state, RETRN to ATN finish code; go through STAAT
ATN Finish	Print structure in STAR if tracing; return to caller of PARSE

2.4.5.2 Code Structures for Tests Here we show the structures compiled for each type of test in the arc condition section. In the case of multiple tests, two consecutive tests will be followed by a reference to AND or OR. A capitalized word shown here means a reference to the appropriate Word.

The Word SKIP causes the IC to be set to the contents of the following word, thus skipping intermediate words. CMPWRD and NEGWRD compare the literal string pointed to by TOS with the string from the lexical unit pointed to by 2OS, and return true or false, respectively. TSTCAT and NEGCAT check that the feature vector pointed to by TOS is a subset of the feature vector in the lexical entry pointed to by 2OS (i.e. that the logical AND of the vectors is equal to the vector pointed to by TOS). [EOS] checks that TOS points to the end-of-sentence lexical unit. LIT puts the following word on TOS.

AD-A072 395 OPERATING SYSTEMS INC WOODLAND CA F/G 9/2
A KNOWLEDGE-BASED AUTOMATED MESSAGE UNDERSTANDING METHODOLOGY F--ETC(U)
JUN 79 G M SILVA, D L DWIGGINS, S G BUSBY F30602-77-C-0144
UNCLASSIFIED RADC-TR-79-133 NL

2 OF 2
AD
A072395

END

WRD (" string" and -" string"):

SKIP	1 word
pointer to LIT word	1 word
number of chars. in string	1 byte
char. string, with padding if needed	n bytes
LIT	1 word
pointer to number byte	1 word
CMPWRD or NEGWRD	1 word

MEM (sequence of WRD tests surrounded by parens):

WRD entry	n words, as above
IFT	1 word
pointer past last WRD entry	1 word
<the above repeated for all but the last WRD entry>	
WRD entry	n words

CAT ([features] and -[features]):

SKIP	1 word
pointer to LIT word	1 word
feature mask	NWRD words
LIT	1 word
pointer to mask	1 word
TSTCAT or NEGCAT	1 word

End Of Sentence ("[EOS]"):

[EOS]	1 word

3.0 Morphology

3.1 General

Ultimately, we would like to implement a fairly general morphographemic processor along the lines of that used in the Kay or Winograd systems. The PATRICIA algorithm for lexical lookup might lend itself well to that process. Also, the chart representation of the sentence would allow us to handle the phrase problem (e.g. whether to represent "heavy bomber" as one lexical unit or two; with a chart, you can do both).

For the current contract, however, we have satisfied ourselves with two lesser extensions to the I&W II approach: first, in lookup, we accept the hyphen as a legal word separator in lexical lookup; i.e. if a lexical entry matches the input string up to the character before a hyphen, we call it a success, create a lexical unit for it, skip the hyphen, and go on to attempt a match for the string following the hyphen.

Second, if a string doesn't match any lexical entry, we use an FSA approach to matching. This will be the subject of the following sections.

3.2 The FSA Matching Process

The idea here is to match the current word (or phrase) in the input against a set of patterns represented as a finite state automaton (FSA). Each final state of the FSA specifies a feature vector in the same manner as a lexical entry. Thus, each pattern found corresponds to a feature vector, and a lexical unit is generated just as in the case of a lexical entry match.

3.3 Syntax of FSA

The syntax of the FSA is quite similar to that of the ATN grammar, a major difference being the form of the tests and the lack of actions on the arcs.

```
fsa ::= 'PATTERNb̸' start-state-name state+ 'b̸ENDPATTERN'

start-state-name ::= state-name

state ::= ':Sb̸' state-name arc+ 'b̸;;'

state-name ::= Word

arc ::= ':Ab̸' test 'b̸=>b̸' state-name 'b̸,,' | ':Fb̸' test feature-set 'b̸,,'

test ::= character-string | *BLANK*

feature-set ::= '[b̸' feature+ 'b̸]'
```

In this syntax, rather than having final states, we have "final arcs", designated by starting with ":F" rather than ":A" as is the case with normal arcs. Thus, when the test succeeds on a final arc, the FSA returns a match indication with a pointer to the associated feature set. The test is simply a list of nonblank characters to be tested against the current character in the string, or the Word *BLANK* to test for the presence of a blank in the string.

As an example, the following FSA will look for numbers, and return the appropriate feature. Assuming four-digit numbers have special significance, it will make a special check for them and return an indication when it finds one.

```
PATTERN NO
:S NO
:A 0123456789 => N1 ,,
;;
:S N1
:A 0123456789 => N2 ,,
:F *BLANK* [ NUM ] ,,
;;
:S N2
:A 0123456789 => N3 ,,
:F *BLANK* [ NUM ] ,,
;;
:S N3
:A 0123456789 => N4 ,,
:F *BLANK* [ NUM ] ,,
;;
:S N4
:A 0123456789 => N5 ,,
:F *BLANK* [ NUM 4DIGIT ] ,,
;;
:S N5
:A 0123456789 => N5 ,,
:F *BLANK* [ NUM ] ,,
;;
ENDPATTERN
```

3.4 FSA Program Structures

The FSA is called via the Word "FSA," which returns either zero for no match, or the pointer to a feature vector. The FSA state structures are much like the ATN state structures, and are defined within a scope block named "FSADF". The arc structures are as follows:

Normal (:A) arcs:
- Arc entry code
- Code for test - leaves truth value on stack
- Code for conditional transfer to specified state or next arc

Final (:F) arcs:
- Arc entry code
- Code for test
- Code for conditional return from FSA with feature vector

3.5 Algorithms for FSA Program Elements

Element	Algorithm
FSA Start	Set FSAPT from LEXWRK, go to first state
State Entry	Go to the first arc
Arc Entry	Set NXTARC to address of next arc
Test Code	Compare character pointed to by FSAPT against specified character(s), return true or false

=>	If TOS true, advance FSAPT, go to specified state; If false, go through NXTARC
[features]	If TOS true, return from FSA with pointer to feature vector; If false, go through NXTARC
End of State	Return failure (=0) from FSA

3.6 Integration with Lexical Lookup

The addition of the FSA pattern matcher to lexical lookup has required some restructuring of the I&W II lookup process. In order to assure that exceptions to patterns can be entered into the lexicon, the FSA is tried only when the current string has failed to match any lexical entry. Also, to allow for future restructuring of the lexicon for more power or greater search efficiency, the Words used in lookup have been redone to provide greater modularity.

The top level remains essentially the same: initiate the sentence chart, loop, performing lookups on each string in turn until the end of the sentence text is reached, then finish the chart. The lookup operation first searches the lexicon, then uses the FSA if lexical lookup failed; if the FSA fails, the current string is checked for period-space, signaling the end of the sentence; if that is not found, the process aborts, signaling an unrecognizable string.

4.0 Event Representation Language Implementation

4.1 General

This section describes the implementation of the Event Representation Language component (ERL) of MATRES II, comprising the template and event record data structures and the search and fill procedures associated with the templates. The formalism which will be used for the abstract specification of both the data structures and the procedures has been adapted from the programming language Prolog, which is described in Attachment II. The advantage of Prolog for our purposes is twofold: first, it is a perspicuous and powerful language for the expression of the concepts of our event representation language, and second, it admits of an effective and reasonably efficient implementation.

We do not attempt to implement the full Prolog programming language; in particular, we only support the basic representation of definite clauses and their evaluation via the basic unification and call/backtrack mechanism. Thus we will refer in the sequel to an implementation of definite clauses rather than Prolog.

The implementation of definite clause procedures described here is an adaptation of the Prolog implementation described in Warren (1977a). Most differences between that implementation and this are due to the difference in machines and system environments; some are simply due to the abandonment of space and time saving features that would have been somewhat expensive to include, or to generalities not required for our application.

4.2 Strategy

Forth, as it exists, does not provide much support for the compilation of languages which differ much in syntax from Forth itself (the ATN language was designed to be very similar to Forth). The semantics of the ERL are such that it could not be represented in a syntax which would be easy to compile with Forth. Therefore, we have taken the following approach.

The syntax of the ERL is that of Prolog, on which it is based. A compiler translates that syntax into Forth Word definitions, which are then loaded with the rest of MATRES II. The compiler itself is a separate program, written in SPITBOL (a dialect of SNOBOL) which, with its recursive pattern-matching facilities and powerful string handling, lends itself well to this task. The compiler will be described in a later section.

The interface between the output of the ATN parser, a tree structure, and the template matching procedures expressed in the ERL is managed by a set of Words which convert the parse tree into a literal representation in the dictionary, as a nested set of skeleton literals, representing the nodes of the tree, and atoms, representing the lexical units at the leaves of the tree. The first compiled ERL predicate is then invoked with the resulting structure as its single argument.

4.3 Data and Procedure Structures

4.3.1 General The major structures in this implementation are the local and global stacks, the trail, and the code area. In the MATRES environment, the code area, including skeletons, resides in the Forth dictionary; for consistency, the parse tree, which is a constant to the semantic interpreter, is copied into this area also.

4.3.2 Environment

4.3.2.1 Stack Structures The local stack is maintained on top of the dictionary (actually, the dictionary pointer is kept pointing above the stack, since the area just above the dictionary is used by the Forth outer interpreter). Local variables are kept on this stack, and consist of "short" (one word) locations, pointing to values in the corresponding frame of the global stack. Thus, the local stack is one word wide. A local frame will have the following fields:

A	Parent goal's argument pointer
X	Parent goal's local frame address
V1	Parent goal's global frame address
TR	Top of trail stack when parent goal was invoked
FL	Failure address (if any) for parent goal
VV	Local frame address for the most recent choice point prior to parent goal
DP	Forth dictionary pointer contents (therefore top of local stack) when parent goal was invoked
NX	Top of global stack (from NXTAVL) when parent goal was invoked.

Thus, the first local variable word is offset 8 words from the frame base. Temporary variables are allocated at the highest offsets.

The global stack resides in the dynamic area on top of the sentence chart (the parse frames are discarded after copying the parse tree), and contains global variables and constructed terms occupying "long" locations (two words wide). Space is maintained on it and addressing performed using the same dynamic space Words as the ATN processor, except for allocation and deletion from the stack top, which is integrally bound up with the clause control mechanisms.

The trail resides in the dynamic area, growing downwards from the top of the dynamic address space, towards the global stack, and contains pointers to variable locations on the local and global stacks.

4.3.2.2 Special Registers The special registers needed to handle the environment are Forth variables. The following registers are used:

V	local stack base for frame being defined during unification; top of local stack after unification.
V1	same for global stack.
W	global stack base for current skeleton being defined during unification; equal to V1 at top level.
VV	image of V for frame associated with last choice point.
VV1	image of V1 for frame associated with last choice point.
TR	pointer to the top of trail stack.
X	local frame base for current goal.
X1	global frame base for current goal.
A	pointer to argument list of current goal.

Y	global frame base for current level goal skeleton; equal to X1 at top level.
B	pointer to argument list of current level skeleton being unified with.

In addition, the dictionary pointer, DP, is used as the pointer to the top of the local stack, and NXTAVL points to the the global stack top.

4.3.2.3 Constructed Terms These terms occupy value cells in the global stack, and are thus long items.

undef	is two zero words.
ref(L)	is a constant zero (0) followed by L, the address of the value cell referenced.
localref(L)	is a constant two (2) followed by L, the address of the local stack word referenced; this only occurs during a pseudo-machine instruction execution, and will be replaced before the end of the instruction.
void	is a constant (4) followed by anything; as with "localref", this only occurs during an instruction.
atom(I)	is a constant six (6) followed by I, the address of the atom literal.
int(I)	is a constant eight (8) followed by I, the value of the integer.
mol(S,F)	is S, the address of the skeleton literal, followed by F, the base address of the associated global frame.

4.3.3 Unification Algorithms

4.3.3.1 Dereferencing Dereferencing is applied to goal arguments to obtain their values. Different algorithms are used for local and global variables. For a global variable, references are followed to return the address of either a value or an **undef** cell. In the latter case, the address is trailed. For a local, if the pointer on the local stack is zero, a new global cell is obtained and filled with a "localref" to the stack word, and the cell's address is stored in the stack word, trailed, and returned; otherwise, the global algorithm is used on the cell pointed to by the local word.

4.3.3.2 Assignment Assignment also differs between global and local variables. For locals, the address of the value cell is placed in the local stack word. For globals, the value itself is placed in the value cell, unless it is **undef**; in that case, the address is turned into a reference and placed in the cell.

4.3.3.3 Trailing Trailing is necessary when an assignment is made to a local word or global cell which is in an environment earlier than the current one, that is when the address is less than the contents of register V (local) or V1 (global). For a local assignment, the address is pushed on the trail stack. For a global, the address of the cell is pushed onto the trail stack after incrementing it by one to indicate that it is a global address. Resetting from the trail consists of getting the address, checking it to see if it is local or global, and setting the global cell to which it points to **undef**, then clearing the local stack word, if any. Both steps are necessary in the local case because there may be other references to the same global cell, and the local word may point to a global cell in a later environment.

In the sequel, the trailing algorithm will be assumed to include the address check for the need to trail.

4.3.3.4 *Unification of Arguments*

		Goal Argument Value				
		Local Var	Global Var	Atom	Integer	Molecule
	Local Var	asg ^ to <	asg ^ to <	asg ^ to <	asg ^ to <	asg ^ to <
	Global Var	asg < to ^, trail	asg ^ to <	asg ^ to <	asg ^ to <	asg ^ to <
Head Arg Type	Atom	asg < to ^, trail	asg < to ^, trail	test if eq	FAIL	FAIL
	Integer	asg < to ^, trail	asg < to ^, trail	FAIL	test if eq	FAIL
	Skeleton	asg < to ^, trail	asg < to ^, trail	FAIL	FAIL	if fncs =, unify args

TABLE 1. Unification Algorithms

The algorithm used to unify an argument of the clause head with the corresponding one in the goal depends on the types of the two. Table 1 gives the algorithms as a function of the types. The type "var" means "a variable which dereferences to **undef**." The notation "^" means the goal argument; ">" means the head argument. Thus "asg ^ to <" means to assign the goal argument to the head argument. The algorithm for void variables is not shown here; nothing is compiled for a void head term, and a void goal term unifies successfully with any head type.

In Table 1, the rows represent the "instructions" to be compiled for each type of argument; that is, the form of an instruction is generally as follows: "dereference the corresponding goal argument, test its type, and perform the appropriate operation".

4.3.4 Compilation of Clauses As in [1], the target language of compilation consists of the "instructions" of a pseudo-machine; these instructions will be described below. In this implementation, they correspond to Forth Words which interpret them.

Each clause is compiled into code for the head clause, a neck instruction, calls and arguments for the body, and a foot instruction at the end. Also, for each predicate of a given arity, there is a procedure consisting of an "enter" followed by "try" calls on each clause in which it appears, in the order of appearance; the last "try" is actually a "trylast".

Unlike [1], skeletons in arguments of both the head and body of a clause are compiled to all levels, and have a dual representation: one, the "goal code", consists of a list of instructions, one for each argument, which allow accessing the values of the arguments. The other, called here the "match code", consists of instructions to do the work of unification. Thus, in general, to unify two skeletons, the match code of one will be executed with the B register set up to point to the goal code of the other, the Y register set up as the global frame base of the goal skeleton, and W set up as the global frame base of the match skeleton

The code for the head of a clause consists of a "head" instruction, followed by match code for the arguments. Similarly, the match code for a skeleton occurring as an argument in a clause consists of match code for the skeleton arguments, followed by a "return". Match code in a top-level body skeleton will only be executed via a call from a "uref" (see below). For example, assume that the clause "member(X,cons(X,L))." is being unified with the goal "member(point(a,b),Ptlist)". The first occurrence of X will unify with the skeleton "point(a,b)", and the second occurrence of X will cause the match code for that skeleton to be called to unify with the first element of "Ptlist".

The following subsections give the instructions used in the goal code and the match code.

4.3.4.1 Goal Code Instruction Definitions These are long items which occur in argument lists of the body or in the goal code of skeletons; the first word of each item contains a parameter, and the second points to a Word which accesses an item of that type. Thus, given a pointer to such an item, the appropriate value may be obtained by the Forth phrase "@+ @ EXEC". During unification, the current vector of goal argument instructions is pointed to by register B.

The value returned by the access Words is always the address of a value cell except in the case of "void". The second word of the item also serves as an indicator of the type of item. The algorithms for the accessing Words are described below.

var(I)	dereferences the Ith variable of the global frame whose base is in register Y; occurs only in goal code. I is compiled as a byte offset from the global frame base.
global(I)	dereferences the Ith variable of the global frame whose base is in register X1; occurs only in argument lists. I is compiled as a byte offset from the global frame base.
local(I)	dereferences the Ith variable of the local frame whose base is in register X; occurs only in argument lists. I is compiled as a byte offset from the local frame base.
void(0)	returns the address of a "void" cell (see **Constructed Terms**) which was initialized at the bottom of the global stack.
atom(I)	I is the address of the atom literal; the access Word creates a value cell for the atom on the global stack and returns its address.
int(I)	I is the value of the integer; the access Word creates a value cell on the global stack containing the integer and returns its address.
[fn(I),...]	The first word of the item is the address of the skeleton literal. The access Word creates a molecule on the global stack consisting of the skeleton address and the contents of the X1 register, and returns the address of the molecule.

4.3.4.2 Match Code Instruction Definitions This section presents the instructions used in the match code for skeletons, and describes the algorithms executed by the instructions. Except as noted, the instructions are actually implemented as Words which take their parameters from the stack, leftmost parameter on TOS. Thus "uskel(N,S)" would show up in the Forth target as "S N USKEL". The variable and argument numbers shown here will actually be adjusted by the compiler to be byte offsets from the appropriate base register values, thus avoiding needless run-time computation.

uskel(N,S)	Argument N of the current skeleton is a skeleton term, and S is the address of the corresponding skeleton literal. Goal argument N at the current level (based on register B) is dereferenced. If the result is a reference, a molecule is formed from S and the contents of the V1 register and stored in the referenced cell; the assignment is trailed. Otherwise, if the result is not a molecule or the functor of the skeleton is different from that of S, failure occurs. If the functors match, registers B, Y, V1 and the address of the next instruction are saved on the local stack, and new values are set for B and Y from the molecule, thus preparing to match the arguments of the goal skeleton against the current (sub)skeleton's arguments. The matching is done by the match code of the skeleton literal at "S".
uvar(N,F,I)	Argument N of the current head skeleton is the first occurrence of variable I of type F (local or global). Argument N of the current goal skeleton is dereferenced and the result is assigned to cell I in the current frame of the "F" stack, unless F is global, the result is a reference and the goal argument is local. In this case, the reference is assigned to the goal variable, and the assignment is trailed. The actual implementation will use two Words, UGVAR for globals and ULVAR for locals, each with two parameters.
uref(N,F,I)	Argument N of the current head skeleton is a subsequent occurrence of variable I of type F (local or global). The value of the variable is obtained, and its type is used to switch to the appropriate unification code. As with "uvar", two Words will be used for the implementation, ULREF and UGREF. In the case where the value is a skeleton, the B, Y, and W registers are pushed onto the local stack, and the match code for the skeleton is executed with B and Y set from the goal molecule and W set from the reference molecule.
uatom(N,I)	Argument N of the current head skeleton is an atom; I is the address of the atom literal. Argument N of the current goal skeleton is dereferenced; if it is a reference, an assignment is made and trailed; otherwise failure occurs unless it is atom "I".
uint(N,I)	Argument N of the current head skeleton is an integer; I is the value of the integer. Argument N of the current goal skeleton is dereferenced; if it is a reference, an assignment is made and trailed; otherwise failure occurs unless it is an integer with value "I".
init(I,J)	Is used to set global variables I through J-1 to **undef**; this must be done just before a "neck" instruction for variables which do not occur in the head, and thus have not been instantiated. Similarly, it must be done before a "uskel" whose skeleton contains first instances of variables, since the skeleton match code may be bypassed. Omitted if no such variables exist.
localinit(I,J)	Like "init", but for local variables; only used before a "neck".

4.3.4.3 Clause and Procedure Control Instructions

enter	is the first instruction in the procedure code for a predicate. It sets up the control information for a new environment by storing the VV, X, A, V1,

	and TR registers, and the dictionary pointer and dynamic stack top (NXTAVL), into the corresponding fields of the local frame, then setting VV and VV1 from V and V1, respectively, to indicate a choice point.
head(I,J)	is the first instruction in the code for the head of a clause having I local variables and J global variables. It checks the two stacks to see that there will be enough room for the variables by adding I+8 words to the Forth dictionary and allocating J cells on the global stack, then sets up the environment for matching arguments by setting registers B and Y equal to registers A and X1, respectively.
neck	precedes the body of a non-unit clause. The success of unification is signaled by setting registers X and X1 from V and V1, and setting V and V1 to the top of their respective stacks.
foot(N)	follows the body of a non-unit clause for a predicate of arity N. If register VV is less than register V, indicating a determinate completion of the current procedure, the dictionary pointer is set from the current DP field and register V is set from register X, thus recovering all local storage used during the procedure. Registers A, X, and X1 are restored from the corresponding fields in the frame based on register X, and control is transferred to the current goal's continuation (after the last goal argument).
neckfoot(N)	follows the head of a unit clause, and is equivalent to "neck, foot(N)", but it takes advantage of the lack of a body. Register V1 is set to the top of the global stack and, if nondeterminate, register V is set to the top of the local stack. The control transfer is then done.
cut(I)	corresponds to an occurrence of the cut symbol in the body of a clause with I local variables (excluding temporaries). Register V is set to the end of the frame based on X (after setting the dictionary pointer from the current DP field) to discard the local frames set up by "neck". Registers VV and VV1 are set from the corresponding fields in the goal frame, based on X, if VV does not already point to an earlier frame than X. Trailed addresses which are greater than VV (i.e. created since the current goal) are removed from the trail.
neckcut(I)	corresponds to a cut occurring as the first goal in the body of a clause. It combines the effect of a "neck" and a "cut" in a straightforward way.
neckcutfoot(N)	corresponds to a cut occurring as the only goal in the body of a clause. It is equivalent to "neck, cut(0), foot(N)". Register V1 is incremented; if nondeterminate, registers VV and VV1 are reset and trail entries are discarded where possible; control is then transferred as with "foot".
return	Is the end of the match code for a skeleton. It restores the B, Y, and W registers from the local stack, and transfers control to the stacked return address.
fehl	corresponds to the goal "fail" in the body of a clause, and causes deep backtracking. Register V is set equal to register VV, and registers V1, A, and X are reset from their corresponding fields based on register V, and X1 is set from the V1 field based on X. The dictionary pointer and NXTAVL are reset from the corresponding fields. Trail entries are popped

	and the words reset until the TR register agrees with the TR field based on V. Note that shallow backtracking, which occurs on a unification failure in the head, is done by just resetting from the trail, unless registers V and VV are unequal, in which case deep backtracking must be done. Both kinds finish by transferring to the address specified in the FL field of the current local frame.
call(L)	corresponds to the the predicate of a goal in the body of a clause. L is the address of the procedure code for the predicate. Register A is set to point to the instruction following the call and control is transferred to L.
try(L)	is used in the procedure code to enter a clause, whose code is at address L. The address of the FL field of the current local frame is set to point to the next location after the "try", and control is transferred to L.
trylast(L)	is used at the end of the procedure code to enter the last clause. Registers VV and VV1 are reset from the corresponding fields of the current local frame, and control is transferred to L.

4.3.4.4 Literals Literals in this interpretation will be Forth Words with compiler-assigned unique Forth names. The name given to the literal in the ERL source (if any) will be stored with the Word as a name string, i.e. in the form returned by WORD. As is usual with Forth, the addresses returned for literals will be the code address, i.e. the address of the first word of the literal data.

A skeleton literal is a Forth array, the first word of which points to the Word identifying the principal functor, the second to the match code, and the third to the goal code.

A functor (or atom) literal is a Forth array, the first word of which is the "arity" of the functor (0 for atoms), and the rest of which is the name string.

4.4 References

[1] David H. Warren, "Implementing PROLOG", volumes 1 and 2, D.A.I. Research Report Nos. 39 and 40, Department of Artificial Intelligence, University of Edinburgh, May 1977

5.0 Event Representation Language Compiler

5.1 Strategy

The compiler is implemented in PDP-11 SPITBOL, using the compiler techniques described in James Gimpel's book "Algorithms in SNOBOL4", Section 18.4, to relate the syntax and semantics. Familiarity with these will be assumed here. The compiler processes a clause at a time, outputting Forth code for the clause and adding to the procedure code for the principal functor of the clause. At the end of the input file, the procedure code for all predicates is output.

Due to space limitations in PDP-11 SPITBOL, the compiler has to be broken into sections; we chose to make two programs: the first pass, which analyzes the syntax and accumulates information on the variables, and the second pass, which produces the code. Pass 1 produces a file called "TEMPLATE.INT", which is read by Pass 2 and processed to produce the final output, called "TEMPLATE.4TH". The input to Pass 1 is called "TEMPLATE.ERL".

5.2 Internal Data Structures

The major data structures in the compiler occur in Pass 2. These are as follows:

IDT	is a table of identifiers of atoms and functors. Each entry in the table is indexed by a string of the form "name arity", and its value is the Forth name by which the atom or functor will be referenced. The table is initialized with names which have "special" Forth names; the usual Forth name is generated by the function TID and is of form "Fn", where "n" is an integer.
PREDT	is a table whose entries are indexed by the Forth name of a functor, and whose values are strings containing the procedure code for the functors.
PREDN	is a table indexed like PREDT, but whose values are the Forth procedure names corresponding to the functor names.
PREDLST	is a list of the Forth functor names of functors encountered during compiling; this is used at the end of compilation to list the predicates whose procedure code is to be output.
VARL	is a list of the variables in a clause; each entry in the list is a string of the form "v <name>" (e.g "vX" for a variable named X). Each such string is the name of a VAR_REC structure containing the number of occurrences of the variable, whether it is global or local, and its offset from the base of its stack.

5.3 Pass 1

The main program consists of a loop which reads and compiles a clause, going to the label EOIN at end-of-file on input, which currently just terminates the program. Each line of the clause is read and any blanks removed (currently a small problem; any blanks in a string literal will also be removed). It is appended to the rest of the clause in string STMT, and the string FS, which was set during input pattern matching, checked. If a terminating period was encountered, it will be in FS, and the clause will be complete. Otherwise, another line is read at RDLP. The completed clause has a blank appended to match the pattern FULL_STOP, and it is output as a comment (this comment was originally to have been passed through Pass 2 into the Forth code, but it caused problems and so was dropped).

The pattern CLAUSE, which contains the syntax of the ERL in SNOBOL-executable form, is passed against the clause twice; first with the variable VAROUT set to output variable information from the semantic routine P1_VAR, and second with CMDOUT set to output semantic routine names and the values of strings found during parsing (variable names, atoms, etc.). After the first pass, a null line is output to delimit the variable information. Each semantic routine in Pass 1 has a corresponding routine of the same name in Pass 2, except for the prefix "Pn_". The semantic routine dispatching code at "S_" outputs the routine names. The semantic routines themselves have only to maintain the term nesting level (to allow labeling variable occurrences as local or global) and to output parsed strings as found and pushed during the pattern match.

5.4 Pass 2

5.4.1 Main Program The main program here reads and discards the line containing the clause text, going to EOIN if an end-of-file is present. Next, the variable information is read, and the list VARL is built and the count of global variables NGLOB and total variables NVAR is made by the loop from VAR to EOVAR. Some complication of this code is caused by the fact that a given occurrence of a global variable may or may not be global.

The null line after the last variable occurrence in the input causes a pattern match failure that causes actual clause compilation to begin. This is done by simply reading the input line and calling the semantic dispatch function "S_", which will transfer to the routine named by the input line. The variable DONE is used as a terminating condition; it is set null at the beginning and non-null by the routine P2_EOCL, which is called at the end of a clause. After compiling the clause, the variable list and each variable name's contents must be deleted to prepare for compiling the next clause; this is done by the loop at EMPTY. Finally, when all clauses have been processed, control comes to EOIN, where the list PREDLST is read and the procedure code output.

5.4.2 Semantic Routines The major semantic routines are covered here. To see the order in which they are called, and what inputs are passed to them, examine the syntax pattern in Pass 1. Each string picked up by a ". PUSH()" there is output by the appropriate routine, and thus is available to a Pass 2 routine by simply reading the input. Routines pass information to each other on the stack via PUSH and POP, usually using the structure FORTH, which has three components: a string comprising the match code for an element, a string comprising the goal code, and a null string unless the element is a variable, in which case it is the variable's offset (this is used by LISTF and SKELF, below).

The routines are listed and described here by name, with the "P2_" prefix omitted.

VAR is triggered by a variable occurrence; the variable name is read and its internal name (i.e. beginning with "v") is assigned to T2. If it is the first occurrence of the variable, OFFSET of the corresopnding VAR_REC has not been set. In this case, we first determine if the variable occurs more than once; if not, it is a void variable, and we go to VDVAR to push the appropriate structure. If so, we set its offset from one of LOFF or GOFF, and update that item (these offsets are initialized by INI, and incremented by the appropriate one of LINC or GINC). Finally, we push the variable FORTH structure, including the just assigned offset.

ATOM is triggered by the occurrence of an atom; it is only necessary to get the atom string, get a Forth name for it via TID, and push the structure for it.

LISTF is called at the end of a list occurrence. The stack will contain the elements of the list, with the top element being the structure for a "nil" atom or whatever term was preceded by ",.."; EL_CT will be set to the number of elements, and NEST will be one greater than its value outside the list. The item HI1 will be set from the offset of the structure at the top of the stack, only if it currently is null; LO1 will be set from the offset only if the offset is not empty. Both here and in SKELF, HI1 and LO1 will be set this way each time a structure is taken from the stack; thus, LO1 will contain the lowest variable offset seen so far, and HI1 the highest (this depends on the fact that the offsets increase from "right to left" in the clause; also, since only global variables occur in these structures, the offsets will be global). The goal and match codes of the top two stack structures are concatenated and output, and a CONS literal is output to reference them. This literal represents the end of the list. Next, a loop indexed by EL_CT is performed, getting the next structure from the stack and building a CONS literal from it and the name of the CONS literal just built. In this way the list is built up as a nest of CONSes. At the end of the loop, a skeleton FORTH structure which contains the name of the top-level CONS is pushed on the stack. If HI1 is not null, and NEST is one, indicating that the list is a top-level argument in the clause, an INIT instruction is prefixed to the match code, and HI1 is set null. This assures that, during unification, variables with initial occurrences in the list will be set to **undef**.

SKELF is called at the end of a complex term occurrence. The item ARGNO will be set to (number of arguments - 1) times 4, this since ARGNO is used during argument processing as the current argument offset. The stack will contain the arguments, last one on top, with the value of ARGNO outside the term below them, and the functor name below that. A loop indexed by the number of arguments is performed, concatenating the match and goal code of the arguments, and setting HI1 and LO1 as described under LISTF. If NEST is zero, indicating that the term is at the top level, the number of arguments and the code structure are pushed back onto the stack to be dealt with by FIN_HEAD (see below). Otherwise, the goal and match code are output followed by a skeleton literal pointing to them, and a code structure for that literal pushed onto the stack (as in LISTF, including an INIT if appropriate).

GOAL handles a term at the top level in the body, and expects what SKELF leaves at NEST level 0. It picks the goal code out of the structure on the stack top, changes occurrences of "VARACCESS" to "GLOBALACCESS", and appends a call to the procedure name of the functor to the clause code (in item CLOUT), with the goal code as the argument list of the call.

5.4.3 Auxiliary Functions

FIN_HEAD is called by routines which occur at the end of a clause head to compile the necessary code. It assumes that the stack contains what SKELF leaves for the top level. It picks up the match code from the top stack element and gets a Forth name for the functor from TID. It gets a new clause name and appends it to CLOUT as the Forth Word name for the clause, then adds a "TRY" of that clause to the procedure code of the functor. Next, a "HEAD" is added to the clause code, using NVAR and NGLOB to compute its arguments, followed by the match code. Finally, "INIT" and "LOCALINIT" instructions are added as necessary (LOFF and GOFF indicate how many variables have occurred in the head, and thus how many have their first occurrence in the body and need to be set to **undef**).

EMIT is used to output Forth code; it is necessary because Forth restricts its input lines to 80 characters. It uses the loop at EMIT_1 along with the function DEC to find the longest string less than 81 characters that is immediately followed by a blank, so that a Word doesn't get split. Thus each string sent to EMIT gets output as one or more lines with no Words split between lines.

TID is used to produce a unique Forth name for atoms and functors. Its arguments are the letter "A" or "F", depending on whether an atom or functor is being input, the source name string, and an integer giving the arity of the functor (or zero for atoms). It concatenates the name string and the arity to make and index to IDT; if there is an entry, it returns its value, which is the Forth name. Otherwise, it makes a new Forth name, enters it into IDT, outputs a "FNLIT" associating the Forth name with the input name string, and (if it is a functor) sets up the procedure for the functor, outputting a "RECURS" definition of the Forth procedure name to be referenced by CALLs, starting the procedure code, and adding the name to PREDLST.

EMITPR is called to output the procedure code for a functor. It converts the last "TRY" in the code to a "TRYLAST"; if this fails, there have been no clauses encountered for this functor, and EMITPR gives up. Otherwise, it outputs the completed code.

5.5 Limitations

This compiler was produced in haste, and had to be "programmed around" some bugs in SPITBOL; thus, it has some shortcomings. As mentioned above, blanks in string literals are rudely squeezed out; also, the semantics will not properly handle an atom (i.e. 0-arity functor) as a top-level term. EMITPR should probably emit a "fail" for a functor with no defining clauses. Since we were dealing with a limited subset of PROLOG, no attempt was made to allow some of the nice features of that language such as number expressions, infix functors, parentheses and semicolons in the body, etc.

6.0 Glossary of Lexical and ATN Words

6.1 Dynamic Storage accessing Words

DYNBAS is a constant - the base block number for pointers to dynamic storage.

NXTAVL is a pointer to the next available byte of storage.

GTNEW (\GTNEW) allocates storage, C(TOS) words (bytes) long (up to 1024 bytes), returns a pointer to the start (will see that storage lies within a FORTH block and update NXTAVL).

TRUNC truncates dynamic storage at the pointer on TOS.

D@ (\D@) returns the word (byte) pointed to by the pointer on TOS.

D@+ (\D@+) operates like D@ (\D@), but returns a pointer to the next word (byte) above the retrieved value.

D+ (\D+) increments the pointer on 2OS by C(TOS) words (bytes).

D! (\D!) stores the word (byte) on 2OS at the location pointed to by TOS.

D!+ (\D!+) operates like D! (\D!), but returns a pointer to the next word (byte).

DOSET sets the word pointed to by TOS to zero.

DDMOVE moves C(TOS) words from the location pointed to by 3OS to that pointed to by 2OS (i.e. "from to length DDMOVE").

DAMOVE (ADMOVE) is like DDMOVE, but moves from (to) dynamic storage to (from) main memory (e.g. dictionary, stack).

\DDMOVE, \DAMOVE, \ADMOVE are byte equivalents of the above.

DTOA produces a main memory address from a dynamic storage pointer (to be used only when absolutely necessary, as when calling a Word which needs a regular address).

6.2 Lexicon Compiler

The working data items are as follows:

FPTR points to the "number of feature vectors" word in the entry currently being defined. This pointer is used to increment the count in the entry.

ENP points to the first word of the entry being defined. The dictionary pointer (DP) is kept pointing to the end of the entry.

LXPTR points to the next available word in the current lexicon block, and is used in moving the newly-defined entry into the block.

FREBYT count of the number of free bytes left in the current lexicon block. This is kept consistent with LXPTR.

CRMASK an NWRD-word array which contains the feature vector (or mask) which is currently being built for the entry. During the definition of the features, it contains the value for the feature next to be defined.

The operational Words in the compiler are as follows:

LEXICON initializes CRMASK for the first feature definition, and starts a scope block for feature Words, to avoid possible conflict between feature names and program names.

FEATURE defines a Word which, when used, will "or" its value into CRMASK. The value of the Word is taken from the current value of CRMASK; CRMASK is then shifted left one bit to form the value for the next feature.

:: (double colon) sets up the next lexical entry, moving in the string for the entry and setting up ENP, FPTR, and pointing DP to the location for the first feature vector.

[clears CRMASK in preparation for building the next feature vector for the entry.

] moves the feature mask from CRMASK to the position pointed to by DP, then updates DP and the count pointed to by FPTR.

.; stores the length of the entry in the count byte of the entry, then moves the entry into the block, starting at the position specified by LXPTR. First, however, it checks FREBYT to see if the entry will fit; if not, it stores an end-of block flag and sets up ELEX, LXPTR, and FREBYT for the next block. In any case, it restores DP to build the next entry in the same space.

ENDLEX stores an end-of-lexicon flag in the location pointed to by LXPTR, terminates the scope block, and flushes the block buffers.

6.3 ATN Processor

6.3.1 Context-Related Words

FRAMBASE is the pointer to the first word of the current state frame. That word in turn contains a pointer to the first word of the previous frame.

REGISTER, LIST are defining words. They define a word as a variable containing the current offset, then increment it. When the defined word is referenced, it will return the then current value of FRAMBASE added to its value. REGISTER defines a one-word cell, LIST a two-word listhead.

CUROF is the current offset to the next defined item in the frame. After all items are defined, it gives the size of the basic state frame.

REGOF is the offset from the frame base to the first register cell. It is used (with CUROF) to delimit the register and list head area.

1STREG is equated to the first defined register offset of the state frame. It is used to identify the start of the register and list head area.

NEWFRAME, OLDFRAME are described elsewhere.

6.3.2 Processor Auxiliary Words

SKPTO, GOTO, SKPS are used to effect transfer of control at the same level of Word invocation (like a "goto", which FORTH doesn't have, but is implicit in the definition of an ATN). The type of transfer depends on whether the address given contains a word in a COLON-type definition or is the code word of a definition. In the case of a code word, the return stack should be left at the level of entry to the parser, so SKPS or SKPTO is used depending on whether the current Word is in the parser or called by a Word in the parser. GOTO is used to transfer within a Word (e.g. to the next arc) at the same level as SKPTO. The actual transfer of control occurs when the next ";" is executed.

LIT is compiled into a definition and, when called, pushes the following word of the definition onto the stack.

\COMP compares two strings of bytes, given TOS=length, 2OS and 3OS having string addresses. Returns 1 if string 2OS < string 3OS, 0 if equal, -1 if string 2OS > string 3OS.

PRSTNM takes a pointer to a state Word parameter address, and prints the Word's name (will actually work for any Word).

DPRSTNM is a conditional version of PRSTNM.

PWORD takes a pointer to a lexical unit, and conditionally prints the string and the address of the lexical unit for tracing.

PRLEVEL corresponds to the current nesting level of the printout of a structure, and is used as the indentation count for a line of printout.

PRDELTA is the amount to increment PRLEVEL for each nesting level.

PTR-TYPE takes a ptr and returns a value depending on the type of element to which it points: 0 - lexical unit, 1 - list, 2 - node.

PRTYP is like TYPE, but indents PRLEVEL spaces first.

PRLEX takes a pointer to a lexical unit and prints the value of the pointer, an ellipsis, and the string for the lexical unit.

PRTPTR takes a ptr and prints the structure to which it points, using PTR-TYPE to select the appropriate printing Word to call.

PRLIST takes a pointer to a list head, prints "LIST OF:", increments PRLEVEL, calls PRTPTR for each listel, decrements PRLEVEL, and prints "END LIST".

PRNODE takes a pointer to a node, prints "NODE:" and the label name, increments PRLEVEL, calls PRTPTR for each branch, decrements PRLEVEL, and prints "END NODE".

FINPARSE types out the structure for the parsed sentence returned in STAR, using PRTPTR.

FALPARSE is called when the parser backtracks out of the initial state, indicating failure of the parse; it simply types a message.

6.3.3 *ATN Processor Words* These are Words which are compiled into the grammar itself.

ARCENT1 is the first Word executed in an arc. It increments ARCNO, prints a message, and calls NEWFRAME, hoping for success.

PSHENT is called at on entry to a PUSH arc; it sets up the first frame for the subnet and clears the registers and lists at that level, but does not make it current; rather it stores its address in NEXTFRAME. This allows items to be sent to the subnet via NEXTFRAME.

ADVLEX takes a pointer to the "new" current lexical unit and pops it into LEX; this is done after the actions of an arc so that LEX refers to the right unit for the actions.

SETSTAR is called at the end of a PUSH arc to reset STAR from LEX.

STPRNT is called on state failure to print out the state name.

STFAIL is called when the conditions on an arc are not met. It types a message, calls OLDFRAME to undo what ARCENT1 did, then sets LEX to the lexical unit pointed to by the alternate pointer of the current unit. If this was the last (or only) alternate, LEX is reset to the first alternate (by finding the next of the previous) and the next arc is made current. In any case, the current arc receives control.

FNDSTATE takes a pointer to a state name in the form required by ?FIND. It calls ?FIND to get the code address of the state; it stores it in STAAT and returns it on the stack. If ?FIND returns zero, however, a message is typed and the parse is aborted.

POPND is the last Word executed for a POP arc. It gets the IC and frame pointer from RETRN, copies the arc address, arc number, state address, and return information from that frame into the current one, copies the registers and list heads from that frame into the current one, pops the TOS value into STAR, and goes to the postactions of the calling PUSH arc through the RETRN IC.

JMPND is the last Word executed for a JUMP arc. It just goes to the state whose name is stored at the beginning of the arc.

1TOEX, 2TOEX are compiled for the TO word of a PUSH arc; 1TOEX takes a pointer to the code word of the state to be "pushed" to and stores it in STAAT, and the frame base of the previous frame in RETRN+2, after moving NEXTFRAME to FRAMBASE, thus setting up the first frame for the subnet; 2TOEX takes the address of the call to it, increments it to be the address of the next word and stores it into RETRN, then goes to the state specified in STAAT.

CMPWRD takes the address of a string of characters on TOS and the address of a lexical unit on 2OS; it returns true if the string matches the string pointed to by the lexical unit.

GTLEX4WRD is called between word items of a MEM list to get the lexical unit pointer, which has been saved on the return stack, back on the parameter stack.

NEGWRDcalls CMPWRD, then reverses the truth value.

TSTCAT takes a pointer to a feature mask on TOS and a pointer to a lexical unit on 2OS; and returns true if the vector pointed to by TOS is a subset of the vector of the lexical unit pointed to by 2OS, i.e. for each bit position of the former containing a one, the corresponding bit position of the latter also contains a one.

NEGCAT calls TSTCAT and reverses the truth value.

ENDSNT tests for the end-of-sentence flag (i.e. a zero in the first word of the lexical unit).

* returns the contents of the special register STAR.

*+1, *-1 return pointers to the next lexical unit and the previous unit, respectively, using the appropriate pointers of the lexical unit pointed to by LEX.

ADV (RET) sets LEX to the next (previous) lexical unit.

SETR simply stores the value in the item.

GETR returns the value of the item.

LABEL is used to declare a node label; it takes a value on the stack which is the number of branches for a node of that type.

NODE makes a node on top of the current frame and returns a pointer to it.

ADDLIST makes a new listel on top of the current frame and links it to the front of the list.

SENDR stores the value into the register location in the next frame.

SENDL moves the list head from the current frame into the next frame.

RETR restores the value of the register into the current frame from the previous one.

RETL restores the value of the list into the current frame from the previous one.

DEFPARSE is used to define PARSE; the variable contents of PARSE are set to point to the initial state of the grammar. PARSE, when called, makes the state name scope block visible, uses NEWFRAME to set up the initial state frame as a pseudo-state, then clears it and fills in selected fields: LEX points to the first lexical unit, RETRN points to FINPARSE and ARC to FALPARSE. Thus, when the top-level POP is done, FINPARSE will receive control; when backtracking out of the initial state, FALPARSE will be the "arc" that is tried. Finally, NEWFRAME is called again to set up the frame for the initial state, and control is transferred to that state.

6.3.4 Compiler Auxiliary Words

NCOND is used during compilation of the conditional part of an arc to count the number of truth values returned by tests which are on the stack at a given point. This is necessary since e.g. a CAT arc may have an additional test. After the conditional it is set to -1 if ADVLEX should be compiled at the end of the actions.

ARCTYPE is set to a number corresponding to the type of arc by the arc entry words. It is used in various places where arc-type-specific code must be compiled.

ARCENT is called by the arc entry words with the arc type on TOS. It stores the arc type in ARCTYPE, compiles a zero and leaves a pointer to it on TOS (this will be used to replace the zero with a pointer to the next arc), compiles a reference to ARCENT1, and zeroes NCOND.

STATFND compiles the next Word in the source into the developing definition preceded by a skip around it and followed by code to get a pointer to the word and call FNDSTATE.

IFT will use the word pointed to by IC as the next IC if TOS is true.

6.3.5 Compiler Words

GRAMMAR stores the next Word in the source stream and puts a pointer to it in PARSE; this will be the name of the first state to be entered by the parser. It also makes the scope block containing the feature Words visible, and starts a scope block for state names.

ENDGRAMMAR makes the feature scope block invisible again and terminates the state name scope block.

:S sets the FORTH compile mode and enters the next word in the source in the FORTH dictionary. The code word for that entry will point to the code following the ";:", which will then be executed upon entry to the state. It prints a message for tracing, zeroes ARCNO, sets ARC to point to the first arc, then transfers control to that arc.

;; compiles the end-of-state code, a pseudo-arc which simply calls STFAIL, prints a message, and then clears the FORTH compile mode.

:WRD, :MEM, :CAT are arc entry Words. They call ARCENT with their type codes and compile a call to "*" to supply an argument for their associated tests.

:TST, :PSH, :POP are arc entry Words like the above, except that they have no implied test, and so do not compile the call to "*". ":PSH" compiles a call to PSHENT.

:JUMP is also an arc entry Word, but, after calling ARCENT, it must enter the following Word from the source, which is the name of the state to jump to, into the dictionary preceded by a skip around it.

!! compiles a logical AND if NCOND is greater than one; it then compiles an IFT which skips around a STFAIL. If the arc type is not a POP or JUMP, which do not advance the lexical pointer, a "*+1" is compiled and NCOND set to -1. If the arc is a PUSH, code is compiled to set SNDP to agree with DP, to prepare for possible SENDRs.

"1 compiles the string following up to the next quote mark into the dictionary preceded by a skip around it and followed by a LIT call to pick up a pointer to the string.

"2 tests MEMFLG; if it's set, compiles an IFT followed by the word on TOS, and puts on TOS the address where that word was stored, then it compiles a call to GTLEX4WRD to restore the lexical unit being tested.

% (-%) compiles a call to "1, CMPWRD (NEGWRD), then "2; then it increments NCOND.

(sets MEMFLG for "2, puts a zero on TOS, then compiles a call to SAVE to save the pointer to the lexical unit to be tested.

) clears MEMFLG, resets the dictionary pointer to remove the last IFT and GTLEX4WRD compiled by "2, then follows the chain of words following IFT calls; in each one, it stores the current dictionary location; finally it increments NCOND and compiles calls to UNSAVE and DROP to remove the saved lexical unit pointer.

AND, OR, NOT compile boolean operations on the results of tests left on the stack. AND and OR also decrement NCOND since they leave one less value on the stack than they find.

[, -[use the Word "[" from the Lexicon Compiler to build the feature vector into CRMASK.

] finishes a CAT test by copying the contents of CRMASK into the dictionary preceded by a skip around it and followed by a LIT call to pick up a pointer to the compiled feature vector; then it compiles a call to TSTCAT or NEGCAT, depending on the value of NEGFLG; then it increments NCOND.

[EOS] compiles a call to ENDSNT, then increments NCOND.

=> compiles an ADVLEX if NCOND is negative; this is done here rather than in "!!" so that LEX points at the current unit during the actions. If the arc is a PUSH type, compiles a call to SETSTAR to make STAR agree with LEX again. STATFND is then called to compile the state finding code, and SKPS is compiled to go to it.

,, compiles the end-of-arc code: POPND for POP arcs, JMPND for JUMP arcs; it then puts the address of the next available dictionary location back into the first word of the arc, using the pointer left on TOS by ARCENT.

TO compiles a call to 1TOEX, then a LIT to pick up the address of the current location in the dictionary, then a call to 2TOEX.

PARSE as defined at the end of the compiler exports the name PARSE out of the ATNDF scope.

6.4 FSA Morphology Words

6.4.1 Morphology Processing Words

FSAARC holds the address of the first word of the current arc being executed.

FSAPT holds the address of the current character being tested.

MATCH is the major Word of the FSA. On entry, LEXWRK holds the address of the first character of the string to be matched. On exit, TOS points to the feature vector for the matched string or zero if no match; FSAPT holds the address of the first character past the matched string, or the character on which the match failed.

ARCENT2 is the code for arc entry; it simply updates FSAARC.

TSTCHAR takes a pointer to a wordstring, and a character pointed to by FSAPT; it returns true if the character is in the string.

TSTFAIL checks TOS; if true, updates FSAPT to the next character position; if false, goes through FSAARC.

6.4.2 Morphology Compiler Words

PATTERN sets the name of the starting state into MATCH and opens the name scope PATDEF

ENDPATTERN closes the name scope PATDEF

:S sets FORTH compile mode, defines the state name as a Word, and establishes state entry code which sets FSAARC and goes to the first arc.

;; compiles a failure return as a last (pseudo-) arc and clears compile mode.

TSTCMPIL compiles code to test current character against string.

ARCENTFSA compiles arc entry code and calls TSTCMPIL to compile test.

:A, :F each calls ARCENTFSA to establish a new arc, and set ARCTYPE to 0 for a non-final arc, 1 for a final arc (could be used for syntax checking, not currently done).

=> compiles a call to TSTFAIL, followed by code to go to the specified state.

[,] compile a call to TSTFAIL, then a call to return the address of the specified feature vector as the result of MATCH.

" compiles the next arc pointer into the first word of the current arc.

6.5 Text Input and Lexical Lookup

The major Words here are GETTXT and MAKESENT, which, when called in turn, will read in the text of a sentence and make the internal representation for it - a "chart" of lexical units.

TXTP is a variable which contains a pointer to the first byte of the sentence text.

TEXSTRT sets up the processing of the sentence text by pointing TXTP and SENTP to where the first character of the text will be stored - the second byte of the next block after the end of the lexicon.

TEXFIN completes the text processing by appending a period set off by blanks to the text and storing the length of the text in the byte just preceding the text.

CHKPUNC classifies the character on TOS - currently, it returns 1 for a comma, 2 for a period, and 0 for any other character.

CHKWRD moves the "word" read by WORD to where SENTP points, preceded by a blank, updates SENTP, then looks for punctuation at the end of the word with CHKPUNC. On finding it, CHKWRD backs up SENTP one character to strip it off, then returns 1 if the punctuation was a period.

GETTXT reads in the text for a sentence using the above words.
The following words are concerned with lexical lookup.

LEXLEN is a constant giving the length of a lexical unit.

STRLEN is a variable containing the length of the string found by lexical lookup.

PVLEX is a variable containing a pointer to the first alternate unit of the most recently found word or phrase.

ALTLEX is a variable containing a pointer to the last-built unit, and is used to link alternates together.

LEXWRK is a variable containing a pointer to the current place in the sentence text for lookup or lexical unit construction.

LEXB is a variable containing the current lexicon block number during lookup.

LEXENTCK takes a pointer to an entry in the lexicon on TOS, and compares its string to the string starting at LEXWRK, using the length of the lexical string as the compare length. If the lexical entry given is really the "end-of-lexicon" entry, it returns -1.

MAK1LEX takes a pointer to a feature vector. It makes a lexical unit, using LEXWRK for the string pointer and STRLEN for the string length, returns a pointer to the new unit, and updates FRAMSTRT.

PVLINK takes a pointer to an old lexical unit and a pointer to a new unit. It puts a pointer to the new unit in the "next" link of each alternate of the old unit, and a pointer to the old unit in the "previous" link of the new.

ALTLINK takes a pointer to an old lexical unit and a pointer to a new unit. It puts the pointer to the new unit in the "alternate" link of the old.

MAKLEXUNT takes a pointer to a lexical entry, and makes the lexical unit(s) corresponding to the entry. It makes the first alternate and links it as the next unit to each alternate of the previous, then makes and links the remaining alternates (if any). It then updates LEXWRK.

MAKMATCHUNT makes a lexical unit for the string found by the FSA, computing the string length from the difference between FSAPT and LEXWRK (less one to account for the terminating delimiter assumed to have been seen by the FSA). Its argument is the feature vector pointer returned from the FSA.

NEXTLEXENT takes a pointer to an entry in the lexicon and returns a pointer to the next entry, advancing to the next block of the lexicon if necessary.

LXLOOKUP searches for a lexical entry to match the text string starting at LEXWRK, using LEXENTCK and NEXTLEXENT. If found, it makes a lexical entry for it using MAKLEXUNT and returns True. If not found, it returns False.

MATCHLOOKUP uses the FSA to attempt a match for the current string. If successful, it makes a lexical entry using MAKMATCHUNT and returns True, otherwise it returns False.

BOMB is used by LOOKUP to print the initial segment of the current string and abort the processing after both the lexical lookup and the FSA fail to match the current string.

LOOKUP attempts to find an interpretation of the current string as a word or phrase by first calling LXLOOKUP and, if that fails, MATCHLOOKUP. If both fail, it checks for the end of the sentence (a period, set off by blanks). If the string is matched and a lexical unit made, LOOKUP returns True. If the end of the sentence is detected, if returns False. If neither, it calls BOMB to inform the operator and terminate processing for the "sentence".

SENSTRT sets up the lexical lookup and lexical unit construction by pointing LEXWRK to the start of the text and makes a dummy lexical unit to start the sentence chart; this is needed by STFAIL to recover the first alternate of the first real unit in the sentence.

SENFIN completes the sentence chart construction by making a lexical unit with a zero string address and length and linking it in; it also sets FRAMSTART to the next available dynamic location.

MAKESENT makes the list of lexical units for the sentence by applying LOOKUP until it reports False.

7.0 Glossary of ERL Target Machine Words

7.1 Trail Management Words

LTRAIL	takes a pointer to a local stack word; makes an entry on the trail if necessary.
GTRAIL	takes a pointer to a global stack cell; makes an entry on the trail if necessary.
EDTRAIL	is called during a "cut" operation to remove from the trail those entries which point into a stack frame which is being discarded.
UNTRAIL	resets trailed locations and pops the trail stack as described in the design document.

7.2 New Global Stack Management Words

GTNEW and its byte equivalent check that the requested space can be used without running into the trail stack; they take and return the same information as their parse namesakes.

7.3 Clause and Procedure Control Instructions

ENTER, HEAD, NECK, FOOT, NECKFOOT, CUT, NECKCUT, NECKCUTFOOT, RETURN, FEHL, FAIL, CALL, TRY, and TRYLAST all implement the instructions as presented in the design document in the straightforward way. In all cases, the first argument is on TOS and the second, if present, is on 2OS. FAIL implements shallow backtracking, and is called only from unification instructions at the instruction level (i.e. the return stack must be at that level). BACKVV does most of a TRYLAST for ERL primitives.

7.4 Dereferencing and Goal Code Words

GDEREF	takes a pointer to a cell on the global stack and dereferences it.
LDEREF	takes a pointer to a word on the local stack and dereferences it, creating a new **undef** cell if necessary.
VARACCESS	is the access Word for skeleton variables.
GLOBALACCESS	is the access Word for global goal variables.
LOCALACCESS	is the access Word for local goal variables.
VOIDACCESS	is the access Word for void variables.
ATOMACCESS	is the access Word for atom instances.
INTACCESS	is the access Word for integers.
MOLACCESS	is the access Word for skeleton arguments.

7.5 Match Code Instructions

GOS	is like a GOTO, but keeps the same execution level.
SELECTOR	is a defining Word for match code Words. It sets up an array of Word addresses to be indexed by goal code index values; thus each Word pointed to handles a particular type of goal argument. When the defined Word is executed, it takes the argument index on TOS and the match parameter(s) on 2OS (etc). When it enters the selected Word, the argument index has been replaced by the address of the value of the corresponding goal item. The Word

	is entered via GOS to maintain the same Forth level of execution; this is necessary since FAIL must be called only at the instruction Word level.
VUSKEL	called to unify a skeleton with a goal variable (local or global). It stores a molecule in the value cell.
CKFUNC	takes two arguments; fails unless both are skeleton literal pointers and both literals have the same predicate - returns nothing. UNSAVE is used to set the return stack to the proper level before the call to FAIL, since CKFUNC is called from the instruction level.
SKUSKEL	takes pointer to value cell on TOS, skeleton literal pointer on 2OS; checks that value is a molecule with same predicate, sets up for argument matching by pushing the return address (the address of the next Word after the call to USKEL), the current values of the B, Y, and W registers, and the address of the next instruction, onto the local stack, then setting B and Y from the molecule. Control is transferred to the match code address taken from the skeleton literal. The stacked values will be restored by the RETURN instruction in the match code. W is saved only for compatibility with the call from 1OSKULREF.
USKEL	is defined via SELECTOR to switch to the appropriate Word for the goal value type to unify it with the skeleton.
ASLVAR	is a Selector Word called to unify a local match variable with a goal value which is an **undef**, atom, integer, or molecule. It stores the address of the value in the match variable, and updates NXTAVL if the value was a new one built by an access Word.
LVLVAR	is a Selector Word called to unify a local match variable with a local goal variable, for which a "localref" has been returned. It changes the "localref" to an **undef** and stores the address of that cell in the match variable.
ULVAR	is the instruction for a local match variable, defined via SELECTOR to switch to the appropriate Word for the goal value type to unify with the variable.
LVGVAR	is a Selector Word called to unify a global match variable with a local goal variable by putting the address of the global variable cell in the local variable word and discarding the localref cell (the dereferencing has already trailed the local variable), and setting the global cell to "undef".
GVGVAR	is a Selector Word called to unify a global match variable with a global goal variable by putting a reference to the goal variable cell in the match variable cell.
ASGVAR	is a Selector Word called to unify a global match variable with an actual value cell by copying the contents of the value cell to the variable cell.
UGVAR	is the instruction for a global match variable, defined via SELECTOR to switch to the appropriate Word for the goal value type to unify with the variable.
LVATOM	is a Selector Word called to unify a match atom with a local goal variable by changing the "localref" cell to an atom cell.

GVATOM is a Selector Word called to unify a match atom with a global variable; it trails the global variable and changes the "undef" cell to an atom cell.

EQATOM is a Selector Word called to unify a match atom with a goal atom by comparing the atom values for equality.

UATOM is the instruction for a match atom, defined via SELECTOR to switch to the appropriate Word for the goal value type to unify with the atom.

LVINT is a Selector Word called to unify a match integer with a local goal variable by changing the "localref" cell to an integer cell.

GVINT is a Selector Word called to unify a match integer with a global variable; it trails the global variable and changes the "undef" cell to an integer cell.

EQINT is a Selector Word called to unify a match integer with a goal integer by comparing the integer values for equality.

UINT is the instruction for a match integer, defined via SELECTOR to switch to the appropriate Word for the goal value type to unify with the integer.

GVUREF is the code to unify a goal with the referenced variable, which has been unified with another variable; it is handled via UGVAR after setting up the stack.

ATUREF is the code to unify a goal with the referenced variable, which has been unified with an atom; it is handled via UATOM after setting up the stack.

INUREF is the code to unify a goal with the referenced variable, which has been unified with an integer; it is handled via UINT after setting up the stack.

0SKUREF is the code to unify a skeleton, which is the value of the referenced variable, with a global goal variable. A molecule is assigned to the goal variable, and the assignment is trailed.

2SKUREF is the code to unify a skeleton, which is the value of the referenced variable, with a local goal variable. A molecule is assigned to the goal variable.

4SKUREF is the code to unify a skeleton, which is the value of the referenced variable, with a void variable. The stack is restored.

SFAIL clears the parameter stack and calls FAIL; it is called when an attempt is made to unify a reference skeleton with an atom or integer.

10SKUREF is the code to unify a skeleton, which is the value of the referenced variable, with a goal skeleton. The functors are compared and, if they are equal, the match code of the referenced skeleton is executed, after saving registers B, Y, and W on the local stack, and setting B and Y from the goal skeleton and W from the frame word of the reference skeleton. The "return" instruction of the match code will restore the registers and return to the level that called "uref".

SSKUREF is a Selector Word to switch to the appropriate code to unify the goal value type with the reference skeleton (the value of the referenced variable).

SKUREF is the code to unify a goal with the referenced variable, which has been unified with a skeleton. It sets up the goal argument index on TOS, and the skeleton and frame words on 2OS and 3OS, and executes SSKUREF to do the work.

REFTYPE is an array of Words used by ULREF to switch to the code to handle different value types.

ULREF is the instruction for a subsequent occurrence of a local match variable. It obtains the value of the variable, and switches to the appropriate Word to handle the value type, leaving a pointer to the value cell on TOS and the argument index on 2OS.

UGREF is the instruction for a subsequent occurrence of global match variable. It uses GDEREF rather than LDEREF to obtain the value of the variable, but subsequently operates just like ULREF, since in both cases the value is a pointer to a global cell containing either a value or **undef**.

INIT takes a global stack offset (in bytes) on TOS and another on 2OS. It clears the words on the stack from that specified on TOS up to that specified on 2OS.

LOCALINIT is like INIT, but works on the local stack.

7.6 ERL Primitives

These Words act like ERL procedures, but operate on "lower-level" data, such as lexical units and the output facilities of FORTH. The ERL compiler recognizes their names and compiles them accordingly.

FEAT takes two arguments, and fails unless the first is a lexical unit, the second is an atom which is the name of a defined feature of the lexicon, and the lexical unit has the specified feature.

TYPATOM takes one argument, and fails if it is not an atom. If it is, it is typed on the console

TYPLEX is like TYPATOM, but requires its argument to be a lexical unit.

TYPCR does a FORTH "CR" function to terminate a line.

LEXEQ takes two arguments, and fails unless the first is a lexical unit, the second is an atom, and the string of the lexical unit is equal to the atom name. This primitive is necessary because ERL-compiled atoms will not unify with lexical units through the normal process.

8.0 Glossary of GLUE File to Connect Parse to Templates

8.1 Strategy

The "glueing" function is performed by first converting the ATN parse tree to an ERL skeleton form in the dictionary. In this representation, lexical units are stored as atoms, with the atom address actually being the lexical unit address. This is unambiguous, since lexical units are stored early in dynamic space, and thus their addresses are small numbers, between the values of SENTP and FRAMSTART. Lists are stored as skeletons representing terms of the form "cons(a,cons(b,...cons(i,nil)...)". Nodes are stored as skeleton literals, the label of the node being a functor literal. It should be noted that the conversion process reverses the elements of a list and the branches of a node; since they are stored in reverse order in the parse tree, this puts them back in the right order for template matching.

Skeleton literals are built by allocating an array in the dictionary for the goal code, and "pushing" the match code onto the dictionary after that. The "build" Words take a pointer to some piece of the parse tree and return a type indicator (0 for atoms, 1 for skeletons) on TOS and a value (the atom address or skeleton literal address) on 2OS.

8.2 Skeleton Building Words

BLLEX	needs only to add the atom indicator to its input.
BLPTR	checks its input; a zero pointer is returned as the atom "nil"; otherwise it calls BLLEX, BLLIST, or BLNODE as indicated by the return from PTR-TYPE.
STPAIR	builds goal and match code for an item returned from a build Word. It takes the goal code address on TOS, the type indicator on 2OS, and the value on 3OS. It stores the match code using "," to update DP, and uses and updates ARGNO to put the argument number in the match instruction. It returns the next available goal address.
BLNODE	builds a skeleton literal for a node. It calls BLPTR to build its branches (and stacks the returns), then sets up the goal code array, sets ARGNO to 0 and calls STPAIR n times to make the match and goal code for the node; finally it builds the three skeleton words on the top of the dictionary and returns their address and the skeleton indicator.
BLLIST	builds a nested set of skeleton literals for a list. It builds each element into a "cons" skeleton, the last one (first one built) having a second argument of "nil". At the entry to the loop, the pointer to the current list element is on TOS, and the return from the last element is on 2OS and 3OS.

8.3 Driver Word

This is the Word used to fully process a sentence; it calls the Word to get the text, do lexical lookup, and build the parse tree; then it calls BLPTR with the contents of STAR to convert the tree. Next it sets up the initial environment for the ERL machine, puts the returned address from BLPTR in the first argument position, and calls P001, the first compiled predicate of the compiled ERL program, assuming that that predicate has a defining clause of the form "name(Tree):-body.".

Appendix A - MATRES II Program Listings

1.0 Top-Level System Module

```
( ================= MATRES.4TH - Top-level Command File ========)
:  ['  135 DELIM ! WORD HERE COUNT TYPE CR ;
@ATNLEX
@LEXICON
@FSA
[' LEXICON LOADED ..]
[' COMPILING GRAMMAR ..]
@GRAMMAR
[' LOADING ERL MACHINE ..]
@ERLM
[' LOADING TEMPLATES ..]
@[60,5]TEMPLATE
@GLUE
[' MATRES READY!]
```

2.0 Lexical and ATN Processing Module

```
       ( See Matres Glossary document for descriptions of these Words)
( we use base) DECIMAL

CODE LIT  S -) IC )+ MOV, NEXT,  ( RETURN CODE ADDRESS; USED TO COMPILE WORDS)

( ======================== Lexicon Compiler ================================ )
80 CONSTANT MAXF     MAXF 16 /MOD SWAP 0> + CONSTANT NWRD   0 VARIABLE NUMF
NWRD ARRAY CRMASK
40 CONSTANT SLEX     SLEX VARIABLE ELEX
0 VARIABLE FPTR      0 VARIABLE ENP         1024 VARIABLE FREBYT
SLEX BLOCK VARIABLE LXPTR

CODE  2*= T S )+ MOV, S ) ROR, T ) ROL, S ) ROL, NEXT,

: FEATURE  IARRAY 0 NWRD 2* CRMASK + CRMASK DO I @ , I 2*= 2 +LOOP DROP
     ' IMMEDIATE EXEC NUMF 1+! ;: CRMASK NWRD MERGE ;

: ::WRD  2 DP +! WORD HERE 1+ \@ 40 = IF 41 DELIM ! WORD THEN
    HERE DUP 1- HERE \@ DUP SAVE 1+ \MOVE UNSAVE ;
: ::  HERE ENP ! ::WRD HERE + 1+ -2 AND DUP OSET DUP FPTR ! 2+ DP ! ;

SC. [ BRACE
: [  CRMASK DUP OSET DUP 2+ NWRD 1- MOVE IMMEDIATE ;
].SC BRACE

: ]  CRMASK HERE NWRD MOVE FPTR @ 1+! NWRD 2* DP +! ;

: .;  ENP @ DUP DP @! SWAP - DUP HERE \! DUP FREBYT @ SWAP - DUP
    0> IF FREBYT ! HERE SWAP LXPTR @ SWAP DUP LXPTR +! 2/ MOVE UPDATE
    ELSE DROP DUP 1024 SWAP - FREBYT ! -1 LXPTR @ ! ELEX @ 1+ DUP ELEX !
    BLOCK OVER OVER + LXPTR ! HERE SWAP ROT 2/ MOVE UPDATE THEN ;

: LEXICON  SC* BRACE [ 1 CRMASK !  [ SC. [ FEATURES SC.V BRACE ] OINT ;

: ENDLEX  LXPTR @ OSET FLUSH [ >.SC FEATURES SC.I BRACE ] OINT
   MAXF NUMF @ < IF [ TOO MANY FEATURES - CHANGE MAXF AND RECOMPILE]
   TYPE CR ABORT! THEN ;


( ================== Dynamic Storage accessing Words ===================== )
0 VARIABLE DYNBAS
0 VARIABLE NXTAVL

: \GTNEW  DUP NXTAVL @ 1023 AND MINUS 1024 + < IF
        NXTAVL DUP @ ROT ROT +!
      ELSE
        NXTAVL @ 1023 + -1024 AND DUP ROT + NXTAVL ! THEN ;
: GTNEW  2* \GTNEW ;

: TRUNC  NXTAVL ! ;

CODE /UMOD  T 2 S I) MOV, 0 CLR, S )+ 0 DIV, S ) T MOV, S -) 0 MOV, NEXT,
: GTAD  1024 /UMOD DYNBAS @ + BLOCK + ;  ( TAKE POINTER, RETURN CORE ADDRESS)
: D@   GTAD @ ;
: \D@  GTAD \@ ;
: D+   LIT 2* , LIT + , IMMEDIATE ;
: \D+  LIT + , IMMEDIATE ;
: D@+   DUP D@ SWAP 2+ ;
: \D@+  DUP \D@ SWAP 1+ ;
: D!  GTAD ! UPDATE ;
: \D!  GTAD \! UPDATE ;
: D!+  SWAP OVER GTAD ! 2+ UPDATE ;
: \D!+  SWAP OVER GTAD \! 1+ UPDATE ;
: DOSET  GTAD OSET ;

: DDMOVE   ROT GTAD ROT GTAD ROT MOVE UPDATE ;
: \DDMOVE  ROT GTAD ROT GTAD ROT \MOVE UPDATE ;
: DAMOVE   ROT GTAD ROT ROT MOVE ;
: \DAMOVE  ROT GTAD ROT ROT \MOVE ;
: ADMOVE   ROT ROT GTAD ROT MOVE UPDATE ;
: \ADMOVE  ROT ROT GTAD ROT \MOVE UPDATE ;

: DTOA  GTAD ;


( =========== Conditional print Words for debugging ====================== )
0 VARIABLE DEBUG
: DBT  DEBUG @ IF TYP1 ELSE 2DROP THEN ;
: DB.  DEBUG @ IF . ELSE DROP THEN ;
: DCR  DEBUG @ IF CR THEN ;
```

```
( =============== Redefinitions of system Words ============================= )
: (.  ' ( EXEC IMMEDIATE ;
: *.  * ;


( ================== State frame definitions ================================= )
0 VARIABLE FRAMSTART   0 VARIABLE SENTP
0 VARIABLE FRAMBASE
2 VARIABLE CUROF

: ITEM  2* CUROF @! DUP CUROF +! VARIABLE ;: @ FRAMBASE @ \D+ ;
: REGISTER  1 ITEM ;   : LIST  2 ITEM ;
( Basic frame items - leave in this order <POPND assumes>)
1 ITEM LEX     1 ITEM STAR    1 ITEM ARC
2 ITEM RETRN   1 ITEM ARCNO   1 ITEM STAAT

CUROF @ VARIABLE REGOF   0 ITEM 1STREG

: NEWFRAME  CUROF @ \GTNEW DUP FRAMBASE @! DUP ROT D!
    1 D+ FRAMBASE @ 1 D+ CUROF @ 2/ 1- DDMOVE ;
: OLDFRAME  FRAMBASE @ DUP TRUNC D@ FRAMBASE ! ;


( ================== ATN Processor Auxiliary Words ======================== )
CODE SKPTO R ) S MOV, S )+ TST, NEXT,
CODE GOTO  R ) S )+ MOV, NEXT,
CODE SKPS  R -) S MOV, S )+ TST, NEXT,

CODE \COMP  0 S )+ MOV, T S )+ MOV, 3 S )+ MOV,
    BEGIN, T )+ \ 3 )+ CMP, NE 1+ , 0 SOB,
    IFGT, T 1 # MOV, ELSE, IFEQ, T CLR, ELSE,
      T -1 # MOV, THEN, THEN, PUSH J,

: PRSTNM  10 - \@+ SWAP CONVERT COUNT TYP1 [ |] TYP1 4 TYP1 ;
: DPRSTNM  DEBUG @ IF PRSTNM ELSE DROP THEN ;

: PWORD  DUP DB. [ .. ] DBT D@+ SWAP GTAD SWAP D@ DBT DCR ;

0 VARIABLE PRLEVEL   2 CONSTANT PRDELTA
: PTR-TYPE  DUP FRAMSTART @ < IF DROP 0 ELSE D@ IF 2 ELSE 1 THEN THEN ;
: PRTYP  CR [ | | | | | | | | | | | | | | | | | | | ] DROP PRLEVEL @ TYP1 TYP1 ;
: PRLEX  DUP CONVERT COUNT PRTYP [ .. ] TYP1 D@+ D@ SWAP GTAD SWAP TYPE ;
RECURS PRTPTR   RECURS PRLIST   RECURS PRNODE
:R PRTPTR  DUP 0= IF [ <<NIL>>] PRTYP DROP ELSE DUP
    PTR-TYPE DUP 0= IF DROP PRLEX ELSE
    1 = IF PRLIST ELSE PRNODE THEN THEN THEN ;
:R PRLIST  [ LIST OF:] PRTYP PRDELTA PRLEVEL +! 1 D+ D@
    BEGIN DUP WHILE D@+ SWAP PRTPTR D@ ENDWHILE DROP
    PRDELTA PRLEVEL -! [ END LIST] PRTYP ;
:R PRNODE  D@+ OVER [ NODE: ] PRTYP PRSTNM PRDELTA PRLEVEL +!
```

```
    DUP ROT @ D+ SWAP DO I D@ PRTPTR 2 +LOOP PRDELTA PRLEVEL -!
    [ END NODE] PRTYP ;

0 VARIABLE PRTREE
: FINPARSE  PRTREE @ IF CR [ PARSE OUTPUT:] TYPE CR STAR D@ PRTPTR THEN
    [ SC.I GRMDF ] OINT ;

CODE FALPARSE  HERE 2+ , 0 , 2 STATE !  [ PARSE FAILED!] TYPE CR ;


( ======================== ATN Processor Words ========================== )
: ARCENT1  ARCNO D@ 1+ DUP ARCNO D! [ TRY ARC # ] DBT DB.
    [ .. ] DBT NEWFRAME ;

0 VARIABLE NEXTBASE
: PSHENT  NEWFRAME 1STREG 0 OVER D! + CUROF @ REGOF @ - \DDMOVE
    FRAMBASE DUP @ D@ SWAP @! NEXTBASE ! FRAMBASE @ D@ NEXTBASE @ D! ;

: ADVLEX  DUP LEX D! STAR D! ;
: SETSTAR  LEX D@ STAR D! ;

: STPRNT  [ STATE ] DBT STAAT D@ 2+ DPRSTNM ;
: STFAIL  [  FAILED!] DBT DCR OLDFRAME
    LEX D@ 4 D+ D@ DUP IF ADVLEX ELSE DROP LEX D@ 3 D+ D@ 2 D+ D@ ADVLEX
        ARC D@ @ ARC D! THEN ARC D@ 2+ GOTO ;

: FNDSTATE  DUP ?FIND DUP 0= IF DROP COUNT TYP1 [ - UNDEFINED STATE]
        TYPE ABORT!
    ELSE SWAP DROP DUP STAAT D! THEN ;

: POPND  FRAMBASE @ D@ NEWFRAME FRAMBASE @ DUP SAVE D! RETRN D@+ D@ FRAMBASE !
    ARC UNSAVE FRAMBASE ! ARC 10 CUROF @ REGOF @ - + \DDMOVE
    SWAP STAR D! GOTO ;

: JMPND  ARC D@ 8 + FNDSTATE SKPTO ;

: 1TOEX  NEXTBASE @ FRAMBASE ! STAAT D! FRAMBASE @ D@ RETRN 1 D+ D! ;
: 2TOEX  4 + RETRN D! STAAT D@ SKPTO ;

: CMPWRD  SWAP D@+ D@ 3 'S @ \@ = IF DTOA SWAP \@+ SWAP \COMP 0=
    ELSE 2DROP 0 THEN ;
: GTLEX4WRD  UNSAVE DUP SAVE ;
: NEGWRD  CMPWRD 0= ;

: TSTCAT  SWAP 5 D+ DTOA DUP NWRD 2* + SWAP
    DO @+ SWAP DUP I @ AND = IF ELSE DROP 0 TERM THEN 2 +LOOP 0 L> ;
: NEGCAT  TSTCAT 0= ;

: ENDSNT  D@ 0= ;
```

```
SC. [ ACTDF
: *  STAR D@ ;     : *+1  LEX D@ 2 D+ D@ ;      : *-1  LEX D@ 3 D+ D@ ;

: ADV  *+1 DUP LEX D! STAR D! ;
: RET  *-1 DUP LEX D! STAR D! ;

: SETR  LIT D! , IMMEDIATE ;
: GETR  LIT D@ , IMMEDIATE ;
: LABEL  WORD HERE DUP \@ SWAP DUP 12 + ROT 1+ \MOVE ENTER ,
    ' DYNBAS 2- @ LAST @ ! HERE \@ 2+ -2 AND DP+! ;
: NODE  DUP @ DUP 2+ GTNEW DUP SAVE SWAP SAVE D! + UNSAVE
    0 DO D! + LOOP DROP UNSAVE ;
: ADDLIST  2 GTNEW DUP ROT 1 D+ GTAD @! ROT ROT D! + D! ;
: SENDR  FRAMBASE @ - NEXTBASE @ + D! ;
: SENDL  DUP D@+ D@ SWAP ROT FRAMBASE @ - NEXTBASE @ + D! + D! ;
: RETR  DUP FRAMBASE @ DUP D@ - - D@ SWAP D! ;
: RETL  DUP FRAMBASE @ DUP D@ - - D@+ D@ SWAP ROT D! + D! ;
].SC ACTDF

: DEFPARSE  VARIABLE ;: NEWFRAME
    FRAMBASE @ DUP DOSET DUP 1 D+ CUROF @ 2/ 1- DDMOVE
    SENTP @ ADVLEX ' FINPARSE RETRN D! ' FALPARSE ARC D! NEWFRAME
    [ SC.V GRMDF ] OINT @ FNDSTATE SKPS ;
0 DEFPARSE PARSE


( ====================== Compiler Auxiliary Words ======================== )
0 VARIABLE NCOND
0 VARIABLE ARCTYPE
: IOINT  STATE @ SAVE STATE 0SET OINT UNSAVE STATE ! ;
: ARCENT  ARCTYPE ! HERE 0 , LIT ARCENT1 , NCOND 0SET [ SC.V TSTWRD ] IOINT ;

: STATFND  LIT SKIP , HERE 0 , DUP WORD 6 DP +! HERE SWAP ! 2÷
    LIT LIT , , LIT FNDSTATE , ;

CODE IFT  S )+ TST, IFEQ, IC )+ TST, NEXT, THEN,
    IC IC ) MOV, NEXT,


( ========================= Compiler Words =============================== )
: GRAMMAR  WORD HERE ' PARSE ! 6 DP +!
    [ SC.V FEATURES SC.V ATNDF SC.V ACTDF SC.[ GRMDF ] OINT ;
: ENDGRAMMAR  [ SC.I FEATURES SC.I ATNDF SC.I ACTDF >.SC GRMDF ] OINT ;

SC. [ ATNDF
SC.V ACTDF
: :S  2 STATE ! CODE ;: DCR [ ENTERING STATE ] DBT DUP DPRSTNM
    [  AT UNIT ] DBT LEX D@ PWORD
    0 ARCNO D! DUP ARC D! 2+ GOTO ;
: ;;  0 , LIT STPRNT , LIT STFAIL , STATE 0SET IMMEDIATE ;
: :WRD   1 ARCENT LIT * , IMMEDIATE ;    : :MEM   2 ARCENT LIT * , IMMEDIATE ;
: :CAT   3 ARCENT LIT * , IMMEDIATE ;
: :TST   4 ARCENT IMMEDIATE ;
: :PSH   5 ARCENT IMMEDIATE LIT PSHENT , ;
: :POP   6 ARCENT IMMEDIATE ;
: :JUMP  7 ARCENT LIT SKIP , HERE 0 , WORD 6 DP +! HERE SWAP ! IMMEDIATE ;

: NOT  0= ;

SC. [ TSTWRD

: !!  NCOND @ 1- 0> IF LIT AND , THEN LIT IFT , HERE 4 + ,
    LIT STFAIL , ARCTYPE @ 5 < IF LIT *+1 , -1 NCOND ! THEN
    [ SC.I TSTWRD ] IOINT IMMEDIATE ;

0 VARIABLE MEMFLG
: "1  LIT SKIP , HERE 0 , DUP 34 DELIM ! WORD HERE \@ 2+ -2 AND DP +!
    HERE SWAP ! 2+ LIT LIT , , ;
: "2  MEMFLG @ IF LIT IFT ,  HERE SWAP , LIT GTLEX4WRD , THEN ;
: "   "1 LIT CMPWRD , "2 NCOND 1+! IMMEDIATE ;
: -"  "1 LIT NEGWRD , "2 NCOND 1+! IMMEDIATE ;

: (  MEMFLG 1SET 0 LIT SAVE , IMMEDIATE ;
: )  MEMFLG 0SET HERE 4 - DUP DP ! DUP
    BEGIN @! DUP WHILE HERE SWAP ENDWHILE DROP
    NCOND 1+! LIT UNSAVE , LIT DROP , IMMEDIATE ;

: AND  NCOND 1-! LIT AND , IMMEDIATE ;
: OR   NCOND 1-! LIT OR , IMMEDIATE ;

0 VARIABLE NEGFLG
: [  SC* BRACE [ IMMEDIATE ;
: -[ SC* BRACE [ NEGFLG 1SET IMMEDIATE ;
: ]  LIT SKIP , HERE DUP 0 , CRMASK HERE NWRD MOVE NWRD 2* DP +!
    HERE SWAP ! 2+ LIT LIT , , NEGFLG @
    IF NEGFLG 0SET LIT NEGCAT ELSE LIT TSTCAT THEN , NCOND 1+! IMMEDIATE ;

: [EOS]  LIT ENDSNT , NCOND 1+! IMMEDIATE ;

].SC TSTWRD

: =>  NCOND @ 0< IF LIT ADVLEX , THEN ARCTYPE @ 5 = IF LIT SETSTAR , THEN
    STATFND LIT SKPS , ' ; , IMMEDIATE ;

: ,,  ARCTYPE @ 6 = IF LIT POPND , ELSE ARCTYPE @ 7 = IF LIT JMPND ,
    THEN THEN HERE SWAP ! IMMEDIATE ;

: TO  STATFND LIT 1TOEX , LIT LIT , HERE , LIT 2TOEX , IMMEDIATE ;

SC.I ACTDF
```

```
].SC ATNDF

: PARSE  SC* ATNDF PARSE ;

( ========================== Text Input =================================== )
0 VARIABLE TXTP
: TEXSTRT  ELEX @ 1+ DYNBAS ! TXTP 1SET SENTP 1SET ;
: TEXFIN  [ . ] SENTP @ SWAP \ADMOVE SENTP @ 2 \D+ DUP 1+ -2 AND DUP SENTP !
    NXTAVL ! TXTP @ - TXTP @ -1 \D+ \D! ;

: CHKPUNC  DUP 44 = IF DROP 1 ELSE 46 = IF 2 ELSE 0 THEN THEN ;
: CHKWRD   BEGIN HERE \@ 0= WHILE RDLN OILN ! OIAD ! WORD ENDWHILE
    HERE DUP 1+ SWAP \@ SENTP @ SWAP \ADMOVE SENTP @ HERE \@  \D+ DUP
    SENTP ! -1 \D+ DUP \D@ CHKPUNC DUP
    IF SWAP SENTP ! 1- IF 1 ELSE 0 THEN ELSE SWAP DROP THEN
    32 SENTP @ \D! + SENTP ! ;

: GETTXT  TEXSTRT BEGIN WORD CHKWRD END TEXFIN ;


( ==================== Morphology Processing Words =========================)
0 VARIABLE FSAARC   0 VARIABLE FSAPT   0 VARIABLE LEXWRK

: DEFMATCH  VARIABLE ;:  LEXWRK @ FSAPT !  [ SC.V PATDEF ] OINT
    @ FNDSTATE SKPS ;
0 DEFMATCH 1MATCH
: MATCH  NEWFRAME 1MATCH [ SC.I PATDEF ] OINT OLDFRAME ;

: ARCENT2  FSAARC @ @ FSAARC ! ;

: TSTCHAR  FSAPT @ \D@ SWAP \@+ DUP ROT + SWAP
    DO DUP I \@ = IF DROP 0 TERM THEN LOOP 0= ;

: TSTFAIL  IF FSAPT 1+! ELSE FSAARC @ 2+ GOTO THEN ;


( ==================== Morphology Compiler Words ===========================)
: PATTERN  WORD HERE ' 1MATCH ! 6 DP +!
    [ SC.[ PATDEF SC.V FSADEF SC.V FEATURES ] OINT ;
: ENDPATTERN  [ >.SC PATDEF SC.I FSADEF SC.I FEATURES ] OINT ;

SC.[ FSADEF
: :S  2 STATE ! CODE IMMEDIATE ;: DUP FSAARC ! 2+ GOTO ;
: ;;  0 , LIT 0 , ' ; , STATE 0SET IMMEDIATE ;

BASE @ OCTAL
: TSTCMPIL  LIT SKIP , HERE 0 , WORD HERE \@
    7 = IF HERE 1+ [ *BLANK*] \COMP 0= IF 20001 DP @ ! THEN THEN
    HERE \@ 2+ -2 AND DP +! HERE OVER ! 2+ LIT LIT , , LIT TSTCHAR , ;
```

```
BASE !
: ARCENTFSA  HERE 0 , LIT ARCENT2 , TSTCMPIL ;

: :A  ARCTYPE 0SET ARCENTFSA IMMEDIATE ;
: :F  ARCTYPE 1SET ARCENTFSA IMMEDIATE ;

: =>  LIT TSTFAIL , STATFND LIT SKPS , ' ; , IMMEDIATE ;

: [  LIT TSTFAIL , SC* BRACE [ IMMEDIATE ;

: ]  LIT SKIP , HERE 0 , CRMASK HERE NWRD MOVE NWRD 2* DP +!
    HERE OVER ! 2+ LIT LIT , , ' ; , IMMEDIATE ;

: ,,  HERE SWAP ! IMMEDIATE ;

].SC FSADEF


( ========================= Lexical Lookup ================================= )
NWRD 5 + CONSTANT LEXLEN ( LENGTH OF LEXICAL ENTRY)
0 VARIABLE STRLEN    0 VARIABLE PVLEX   0 VARIABLE ALTLEX
0 VARIABLE LEXB

: LEXENTCK  \@+ SWAP
    IF \@+ LEXWRK @ DTOA ROT OVER OVER + \@ DUP 32 = SWAP 45 = OR DUP
      IF DROP \COMP 0= ELSE SAVE 2DROP DROP UNSAVE THEN
    ELSE DROP -1 THEN ;

: MAK1LEX  0 PVLEX @ 0 STRLEN @ LEXWRK @ LEXLEN GTNEW DUP SAVE
     D! + D! + D! + D! + D! + NWRD ADMOVE UNSAVE DUP PWORD ;
: PVLINK  OVER OVER SWAP 3 D+ D!
    BEGIN DUP WHILE OVER SWAP 2 D+ D! + 1 D+ D@ ENDWHILE 2DROP ;
: ALTLINK  4 D+ D! ;
: MAKLEXUNT  1+ \@+ OVER STRLEN ! + 1+ -2 AND
    @+ DUP MAK1LEX SWAP OVER DUP ALTLEX ! PVLEX @ PVLINK ROT
    1- DUP IF 0 DO NWRD 2* + DUP MAK1LEX DUP ALTLEX @! ALTLINK LOOP DROP
    ELSE 2DROP THEN PVLEX ! STRLEN @ 1+ LEXWRK +! ;
: MAKMATCHUNT  FSAPT @ DUP LEXWRK @  - 1- STRLEN !
    SWAP MAK1LEX DUP PVLEX @! PVLINK LEXWRK ! ;

: NEXTLEXENT  DUP \@ + DUP \@ 255 =
    IF DROP LEXB @ 1+ DUP LEXB ! BLOCK THEN ;

: LXLOOKUP  SLEX DUP LEXB ! BLOCK
    BEGIN DUP LEXENTCK 0= WHILE NEXTLEXENT ENDWHILE
    DUP \@ IF MAKLEXUNT 1 ELSE DROP 0 THEN ;

: MATCHLOOKUP  MATCH DUP IF MAKMATCHUNT 1 THEN ;

: BOMB  CR LEXWRK @ GTAD 10 TYPE ABORT! ;
: LOOKUP  LXLOOKUP
    IF 1
    ELSE MATCHLOOKUP
      IF 1
      ELSE LEXWRK @ GTAD [ . ] \COMP DUP
        IF [ UNRECOGNIZABLE STRING! ] TYPE BOMB THEN
      THEN
    THEN ;

: SENSTRT  TXTP @ LEXWRK ! SENTP @ PVLEX !
    0 0 0 0 5 GTNEW D! + D! + DUP 3 D+ SWAP D! + D! + D! + SENTP ! ;
: SENFIN  STRLEN 0SET LEXWRK 0SET SC* BRACE [ CRMASK MAK1LEX DUP
    PVLEX @ PVLINK DUP 2 D+ D! NXTAVL @ FRAMSTART ! ;
: MAKESENT  SENSTRT BEGIN LOOKUP WHILE ENDWHILE SENFIN ;


( =========================== Utility Words ================================= )
: >.SC  ].SC ; ( THIS TO FAKE "[", WHICH LOOKS FOR "]")

: DDUMP  GTAD SWAP GTAD SWAP DUMP ;

: \DDUMP  GTAD SWAP GTAD SWAP \DUMP ;

0 DEFPARSE SUBPARS
: SUBPARSE  WORD HERE ' SUBPARS ! 6 DP +! SUBPARS 6 DP -! ;

: FRMDMP  FRAMBASE @ DUP CUROF @ + DDUMP ;

: >>  GETTXT MAKESENT PARSE ;
: >>>  GETTXT MAKESENT SUBPARSE ;
```

3.0 ERL Machine

```
( ========================== ERL Pseudo-Machine =============================)

( Miscellany)
: DP!  DP ! ;
0 VARIABLE ETRACE
CODE SKPS  IC R )+ MOV, V S )+ MOV, V @)+ JMP,
CODE SKPTO  R )+ TST, ' SKPS P JMP,

: [.  STATE 0SET IMMEDIATE ;  ( LEAVE COMPILE MODE IN DEF.)
: .]  2 STATE ! ;  ( REENTER COMPILE MODE IN DEF.)
: [:  2 STATE ! ;  ( ENTER COMPILE MODE TO STORE CODE)
: :]  STATE 0SET IMMEDIATE ; ( LEAVE COMPILE MODE)

8 C 8  12 C 12  16 C 16  18 C 18  20 C 20  22 C 22 24 C 24

( Special Registers)

0 VARIABLE V    0 VARIABLE V1    0 VARIABLE VV    0 VARIABLE VV1
```

```
0 VARIABLE X     0 VARIABLE X1    0 VARIABLE TR    0 VARIABLE A
0 VARIABLE B     0 VARIABLE Y     0 VARIABLE W


( P_TERM takes a pointer to a value cell, prints the contents)
RECURS P_TERM
: P_ATOM  1 D+ D@ DUP FRAMSTART @ L< IF D@+ SWAP GTAD SWAP D@ TYP1
    ELSE 2+ COUNT TYP1 THEN ;
: P_INT  1 D+ D@ CONVERT COUNT TYP1 ;
: P_GVAR  [ G_] TYP1 CONVERT COUNT TYP1 ;
: P_LVAR  [ L_] TYP1 CONVERT COUNT TYP1 ;
: P_VOID  DROP [ <VOID>] TYP1 ;
: P_SKEL  Y @ SWAP D@+ D@ Y !
    @+ SWAP @+ COUNT TYP1 SWAP 2+ @ SWAP [ (] ROT
    0 DO TYP1 @+ @+ ROT ROT EXEC P_TERM [ ,] LOOP
    2DROP DROP [ )] TYP1 Y ! ;
IARRAY P_TYPE  ' P_GVAR , ' P_LVAR , ' P_VOID , ' P_ATOM ,
               ' P_INT , ' P_SKEL ,
:R P_TERM  NXTAVL @ SWAP DUP D@ DUP 8 L> IF DROP 10 THEN
    P_TYPE + @ EXEC NXTAVL ! ;
0 VARIABLE P_SW
: P_TERM  P_SW @ IF CR P_TERM ELSE DROP THEN ;


( Functor Literals)

: FNLIT  CONSTANT 96 DELIM ! WORD HERE \@+ + 5 + -2 AND DP! ;
0 FNLIT NIL nil
2 FNLIT CONS cons


( Field Definitions in Local Stack)

: FIELD  CONSTANT ;: @ V @ + ;
 0 FIELD AF    2  FIELD XF    4 FIELD V1F   6 FIELD TRF
 8 FIELD FLF  10 FIELD VVF   12 FIELD DPF  14 FIELD NXF


( Trail Management Words)
: TRPUSH  TR @ -1 D+ DUP TR ! D! ;
: LTRAIL  DUP V @ L< IF TRPUSH ELSE DROP THEN ;
: GTRAIL  DUP V1 L< IF 1+ TRPUSH ELSE DROP THEN ;

: EDTRAIL  VV @ TRF @ TR @ OVER OVER -
    IF DO I D@ OVER OVER L> IF DROP ELSE DOSET THEN 2 +LOOP
    ELSE 2DROP THEN DROP ;

: UNTRAIL  TRF @ TR @ OVER OVER -
    IF DO I D@ DUP IF
```

```
     DUP 1 AND IF 0 0 ROT 1- D!+ D! ELSE DUP @ 0 0 ROT D!+ D! 0SET THEN
     ELSE DROP THEN 2 +LOOP
   ELSE 2DROP THEN TRF @ TR ! ;


( New Global Stack Management Words)
: \GTNEW  NXTAVL @ DUP ROT + DUP NXTAVL !
   TR @ L>= IF [ GLOBAL STACK OVERFLOW!] TYPE ABORT! THEN ;
: GTNEW  2* \GTNEW ;


( Clause and Procedure Control Instructions)

: GOS GOTO ;

: ENTER  VV @ VVF ! X @ XF ! A @ AF ! V1 @ V1F ! TR @ TRF !
   HERE DPF ! NXTAVL @ NXF ! V @ VV ! V1 @ VV1 ! ;

: HEAD  V @ + DP! V1 @ + DUP NXTAVL @ - DUP 0> IF \GTNEW DROP NXTAVL !
   ELSE 2DROP THEN A @ B ! X1 @ Y ! V1 @ W ! ;

: NECK  V @ X ! V1 @ X1 ! HERE V ! NXTAVL @ V1 ! ;

: FOOT  VV @ X @ L< IF X @ DUP V ! DP! THEN V @ X @ V !
   AF @ A ! XF @ DUP X ! V ! V1F @ X1 ! V ! A @ + GOTO ;

: NECKFOOT  NXTAVL @ V1 ! VV @ V @ = IF HERE V ! ELSE V @ DP! THEN A @ + GOTO ;

: CUT  X @ DUP V ! VVF @ DUP V ! VV ! V1F @ VV1 ! EDTRAIL + DUP V ! DP ! ;

: NECKCUT  V @ X ! V1 @ X1 ! V +! V @ DP ! V1 +!
   V @ X @ V ! VVF @ DUP V ! VV ! V1F @ VV1 ! V ! EDTRAIL ;

: NECKCUTFOOT  NXTAVL @ V1 ! V @ VV @ =
   IF V @ VVF @ DUP VV ! V ! V1F @ VV1 ! V ! EDTRAIL THEN A @ + GOTO ;

: RETURN  HERE 8 - DUP DP ! @+ @+ @+ @ V1 ! Y ! B ! GOTO ;

: FEHL  VV @ V ! V1F @ V1 ! AF @ A ! XF @ DUP X ! ' V1F @ + @ X1 ! DPF @ DP !
   NXF @ TRUNC UNTRAIL ;
: FAIL  VV @ V @ <> IF FEHL ELSE UNTRAIL THEN UNSAVE DROP FLF @ GOTO ;
: FEHL  FEHL UNSAVE DROP FLF @ GOTO ;

: CALL  UNSAVE DUP SAVE A ! SKPTO ;

0 VARIABLE PTRY
: TRY  PTRY @ IF CR [ TRY] TYPE DUP PRSTNM THEN UNSAVE DUP SAVE FLF ! GOTO ;

: BACKVV  V @ VVF @ DUP VV ! V ! V1F @ VV1 ! V ! ;
: TRYLAST  PTRY @ IF CR [ TRYLAST] TYPE DUP PRSTNM THEN BACKVV GOTO ;
```

```
( Dereferencing Words)

RECURS GDEREF
:R GDEREF  DUP D@ 0= IF DUP 1 D+ D@ DUP 0= IF DROP
    ELSE SWAP DROP GDEREF THEN THEN ;

: LDEREF  DUP @ 0= IF DUP 2 GTNEW DUP ROT 2 ROT D! + D! DUP ROT DUP LTRAIL !
    ELSE @ GDEREF THEN ;


( Access Words)

: VARACCESS  Y @ + GDEREF ;
: GLOBALACCESS  X1 @ + GDEREF ;
: LOCALACCESS  X @ + LDEREF ;
: VOIDACCESS  DROP NXTAVL @ 4 OVER D! ;
: ATOMACCESS  2 GTNEW DUP ROT 6 ROT D! + D! ;
: INTACCESS  2 GTNEW DUP ROT 8 ROT D! + D! ;
: MOLACCESS  2 GTNEW DUP ROT Y @ SWAP ROT D! + D! ;


( Match Code Instructions)

: SELECTOR  CODE  ;: SWAP B @ + @+ @ EXEC DUP P_TERM
    DUP D@ DUP 8 L> IF DROP 10 THEN ROT + @ GOS ;

: VUSKEL  DUP GTRAIL D! + W @ SWAP D! ;
: DFAIL  2DROP FAIL ;
: CKFUNC  DUP 10 L> IF @ ELSE 2DROP UNSAVE DROP FAIL THEN  SWAP
          DUP 10 L> IF @ ELSE 2DROP UNSAVE DROP FAIL THEN = IF
        ELSE UNSAVE DROP FAIL THEN ;
: SKUSKEL  OVER OVER D@ CKFUNC UNSAVE DUP SAVE , B @ , Y @ , W @ ,
    D@+ D@ Y ! 4 + @ B ! 2+ @ GOTO ;
SELECTOR USKEL  ' VUSKEL DUP , , ' DROP , ' DFAIL , ' DFAIL ,
    ' SKUSKEL ,

: ASLVAR  SWAP V @ + ! ;
: LVLVAR  DUP 0 0 ROT D! + D! ASLVAR ;
SELECTOR ULVAR  ' ASLVAR , ' LVLVAR , ' DROP , ' ASLVAR DUP DUP , , ,

: LVGVAR  SWAP W @ + DUP ROT 1 D+ D@ ! 0 0 ROT D! + D! 4 NXTAVL -! ;
: GVGVAR  0 ROT W @ + D! + D! ;
: ASGVAR  SWAP W @ + DUP GTRAIL 2 DDMOVE ;
SELECTOR UGVAR  ' GVGVAR , ' LVGVAR , ' DROP , ' ASGVAR DUP DUP , , ,

: LVATOM  6 SWAP D! + D! ;
: GVATOM  DUP GTRAIL LVATOM ;
: EQATOM  1 D+ D@ <> IF FAIL THEN ;
SELECTOR  UATOM  ' GVATOM  , ' LVATOM , ' DROP , ' EQATOM , ' DFAIL DUP , ,
: LVINT  8 SWAP D! + D! ;
: GVINT  DUP GTRAIL LVINT ;
: EQINT  1 D+ D@ <> IF FAIL THEN ;
SELECTOR  UINT  ' GVINT , ' LVINT , ' DROP , ' DFAIL , ' EQINT , ' FAIL ,

: GVUREF  W @ - SWAP LIT UGVAR SKPS ;
: ATUREF  1 D+ D@ SWAP LIT UATOM SKPS ;
: INUREF  1 D+ D@ SWAP LIT UINT SKPS ;
: 0SKUREF  DUP GTRAIL D! + D! ;
: 2SKUREF  D! + D! ;   : 4SKUREF  2DROP DROP ;   : SFAIL  DROP 2DROP FAIL ;
: 10SKUREF  OVER OVER D@ CKFUNC UNSAVE DUP SAVE , B @ , Y @ , W @ ,
    D@+ D@ Y ! 4 + @ B ! SWAP W ! 2 + @ GOTO ;
SELECTOR SSKUREF  ' 0SKUREF , ' 2SKUREF , ' 4SKUREF , ' SFAIL DUP , ,
    ' 10SKUREF ,
: SKUREF  D@+ D@ SWAP ROT LIT SSKUREF SKPS ;
IARRAY REFTYPE  ' GVUREF , 0 , 0 , ' ATUREF ,
    ' INUREF , ' SKUREF ,
: ULREF  SWAP V @ + LDEREF DUP D@ DUP 8 L> IF DROP 10 THEN REFTYPE + @ GOS ;

: UGREF  SWAP W @ + GDEREF DUP D@ DUP 8 L> IF DROP 10 THEN REFTYPE + @ GOS ;

: INIT  W @ DUP SAVE + SWAP UNSAVE + SWAP DO I DOSET 2 +LOOP ;
: LOCALINIT  V @ DUP SAVE + SWAP UNSAVE + SWAP DO I OSET 2 +LOOP ;


( Primitive predicates of ERL)

: FFAIL  FAIL ;
: FEAT  ENTER BACKVV 0 16 HEAD B @ @+ @ EXEC D@+ D@ SWAP
    6 <> IF DROP FFAIL THEN DUP FRAMSTART @ L> IF DROP FFAIL THEN
    DUP SENTP @ L< IF DROP FFAIL THEN
    B @ 4 + @+ @ EXEC D@+ D@ SWAP 6 <> IF DROP FFAIL THEN
    2+ [ SC.V FEATURES ] OINT ?FIND [ SC.I FEATURES ] OINT
    DUP 0= IF DROP FFAIL THEN 2+ TSTCAT IF 8 NECKFOOT ELSE FFAIL THEN ;

: TYPATOM  ENTER BACKVV 0 16 HEAD B @ @+ @ EXEC D@+ D@ SWAP
    6 <> IF DROP FFAIL THEN
    DUP FRAMSTART @ L> IF 2+ COUNT TYPE ELSE
    D@+ SWAP GTAD SWAP D@ TYPE THEN 4 NECKFOOT ;

: TYPLEX  ENTER BACKVV 0 16 HEAD B @ @+ @ EXEC D@+ D@ SWAP
    6 <> IF DROP FFAIL THEN
    DUP FRAMSTART @ L> IF DROP FFAIL ELSE
    D@+ SWAP GTAD SWAP D@ TYPE THEN 4 NECKFOOT ;

: TYPCR  ENTER BACKVV 0 16 HEAD CR 4 NECKFOOT ;

: LEXEQ  ENTER BACKVV 0 16 HEAD B @ @+ @ EXEC D@+ D@ SWAP
    6 <> IF DROP FFAIL THEN DUP FRAMSTART @ L> IF DROP FFAIL THEN
    DUP SENTP @ L< IF DROP FFAIL THEN
    B @ 4 + @+ @ EXEC D@+ D@ SWAP 6 <> IF DROP FFAIL THEN
    2+ CMPWRD IF 8 NECKFOOT ELSE FFAIL THEN ;
```

4.0 ERL Compiler - Pass 1

```
*****************************************************************************
*                                                                           *
*                   EVENT REPRESENTATION LANGUAGE COMPILER                  *
*   PASS 1 - OUTPUT VARIABLE INFO AND INSTRUCTION, DATA SEQUENCE            *
*                                                                           *
*****************************************************************************
************************ DATA DEFINITIONS ***********************************
*
  DATA('LINK(NEXT,VALUE)');   DATA('VAR_REC(NOCC,TYPE,OFFSET)')
*
************************* INITIALIZATIONS **********************************
*
  INPUT(.INSTR,1,'SY:TEMPLATE.ERL')
  OUTPUT(.OUTSTR,2,'SY:TEMPLATE.INT')
  OUTPUT(.CONSOL,7)
  &ANCHOR = 1
  LINC = 2; GINC = 4
  SETEXIT('ERR');   &ERRLIMIT = 1;   &STLIMIT = -1
*
*********************** FUNCTION DEFINITIONS *******************************
*    DEFINES
  DEFINE('OPT(PATTERN)');
  DEFINE('S(NAME)');   DEFINE('S_(NAME)T1,T2,T3,T4')
  DEFINE('PUSH(X)');   DEFINE('POP()');   DEFINE('TOP()')
  DEFINE('OSPAN(PAT)');
  :(ENDEF)
*
*    DEFINITIONS
ERR   DUMP(2)   :(ABORT)

OSPAN
  OSPAN = SPAN(PAT) | NULL   :(RETURN)

PUSH
  PUSH_POP = LINK(PUSH_POP,X);   PUSH = .VALUE(PUSH_POP)   :(NRETURN)

POP
  IDENT(PUSH_POP)   :S(FRETURN)
  POP = VALUE(PUSH_POP);   PUSH_POP = NEXT(PUSH_POP)   :(RETURN)

TOP
  TOP = DIFFER(PUSH_POP) .VALUE(PUSH_POP)   :F(FRETURN) S(NRETURN)

OPT
  OPT = PATTERN | NULL   :(RETURN)
```

```
************************* SEMANTIC ROUTINES ****************************
*   MAIN ROUTINE DEFINITIONS
S
  S = EVAL("NULL . *S_(." NAME ")")   :(RETURN)
S_  S_ = .DUMMY;  CMDOUT = NAME  :($('P1_' NAME))

*   SEMANTIC ROUTINES
*
P1_VAR
  T1 = POP();  CMDOUT = T1
  VAROUT = T1 ':' (LE(NEST,1) '','G')
  :(NRETURN)

P1_VDVAR  :(NRETURN)

P1_ATOM  CMDOUT = POP()  :(NRETURN)

P1_INT  CMDOUT = POP()   :(NRETURN)

P1_CONSI  :(NRETURN)

P1_INCL  :(NRETURN)

P1_NIL  :(NRETURN)

P1_LISTI
  NEST = NEST + 1
  :(NRETURN)

P1_LISTF
  NEST = NEST - 1
  :(NRETURN)

P1_INC  :(NRETURN)

P1_SKELI
  NEST = NEST + 1;  CMDOUT = POP()
  :(NRETURN)

P1_SKELF
  NEST = NEST - 1
  :(NRETURN)

P1_GOAL :(NRETURN)

P1_CNEK  :(NRETURN)

P1_CFOOT  :(NRETURN)
```

```
P1_CNC  : (NRETURN)

P1_CNCF  : (NRETURN)

P1_CNF  : (NRETURN)

P1_EOCL  : (NRETURN)

P1_INI
  PUSH_POP =  ;  NEST = 0
  : (NRETURN)
*
ENDEF
*
*************** SYNTAX FOR ERL - MAIN SYMBOL "CLAUSE" **********************
*
 PUNCTUATION_CHAR = '()[]{},'
 SOLO_CHAR        = ';!|'
 SYMBOL_CHAR      = '+-*/\^<>=':.?@#$%&'
 &ALPHABET  TAB(32) LEN(96) . UNQALPHA
 UNQALPHA  ARB '"'    =
 UNQALPHA  ARB "'"    =
 DIGITS           = '0123456789'
 LCLETTERS        = 'abcdefghijklmnopqrstuvwxyz'
 UCLETTERS        = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
 ALPHANUMERICS    = UCLETTERS LCLETTERS DIGITS '_'
 NUMBER           = ANY(DIGITS) OSPAN(DIGITS)
 SYMBOL           = ANY(SYMBOL_CHAR) OSPAN(SYMBOL_CHAR)
 WORD             = ANY(LCLETTERS) OSPAN(ALPHANUMERICS)
 QUOTED_NAME      =   "'" SPAN(UNQALPHA '"') . *PUSH() "'"
+                   | '"' SPAN(UNQALPHA "'") . *PUSH() '"'

 FULL_STOP        = '. '
 VARIABLE         = (ANY(UCLETTERS) OSPAN(ALPHANUMERICS)) . *PUSH() S(.VAR)
+                   | '_' S(.VDVAR)
 INTEGER          =  (NUMBER | '-' NUMBER) . *PUSH() S(.INT)
 ATOM             =   QUOTED_NAME
+                   | (  WORD
+                      | SYMBOL
+                      | SOLO_CHAR
+                     ) . *PUSH()
 LISTEXPR         = *TERM ARBNO(',' S(.INCL) *TERM)
+                   (  ',..' S(.CONSI) *TERM
+                    | NULL S(.CONSI) S(.NIL)
+                   )
 LIST             =  '[]' S(.NIL)
+                   | '[' S(.LISTI)
+                     LISTEXPR
+                     ']' S(.LISTF)
 ARGUMENTS        = *TERM ARBNO(',' S(.INC) *TERM)
 TERM              =  ATOM
+                      ( '(' S(.SKELI) ARGUMENTS ')' S(.SKELF)
+                       | NULL S(.ATOM)
+                      )
+                     | VARIABLE
+                     | INTEGER
+                     | LIST
 GOAL              =  (TERM | '!' . *PUSH() S(.ATOM)) S(.GOAL)
 GOALS             =  GOAL ARBNO(',' GOAL)
+                     OPT(',' *GOALS)
 HEAD              = TERM
 CLAUSE            = S(.INI) HEAD
+                     (  ':-' S(.CNEK) GOALS S(.CFOOT)
+                      | ':-!'
+                        (  ',' S(.CNC) GOALS S(.CFOOT)
+                         | NULL S(.CNCF)
+                        )
+                      | NULL S(.CNF)
+                     ) FULL_STOP S(.EOCL)
*
**************************************************************************
*                               MAIN PROGRAM
**************************************************************************
** READ AND COMPILE A CLAUSE
CMPL
  STMT =
*   READ A LINE AND ADD IT TO STRING "STMT"
RDLP
  TRIM(INSTR)  OSPAN(' ') ARB  . LINE ('.' | NULL) . FS RPOS(0)  :F(EOIN)
*     UNLESS IT'S A COMMENT LINE
  LINE  ANY('|/*')  :S(RDLP)
*   SQUEEZE OUT BLANKS
  &ANCHOR = 0
SQZ  LINE  ' ' =   :S(SQZ)
  &ANCHOR = 1
  STMT = STMT LINE FS
  IDENT(FS)  :S(RDLP)
*   A PERIOD AT THE END OF LINE TERMINATES A CLAUSE
  STMT = STMT ' '
*   PUT THE SOURCE IN THE OBJECT AS A COMMENT
  OUTSTR = '( ' REPLACE(STMT,'()','{}') ')'
*   PARSE AND OUTPUT THE VARIABLE INFO FOLLOWED BY A NULL LINE
  OUTPUT(.VAROUT,2)
  STMT  CLAUSE   :F(ERROR)
  VAROUT =
  DETACH(.VAROUT)
*   PARSE AND OUTPUT THE COMMAND AND DATA STUFF
  OUTPUT(.CMDOUT,2)
  STMT  CLAUSE   :F(ERROR)
  DETACH(.CMDOUT)  :(CMPL)
```

```
*   END OF INPUT
EOIN
  : (END)
ERROR
  CONSOL = STMT
  OUTSTR = '****** SYNTAX ERROR IN ABOVE CLAUSE **********'
  CONSOL = OUTSTR
  : (CMPL)
END
```

5.0 ERL Compiler - Pass 2

```
*****************************************************************************
*
*                   EVENT REPRESENTATION LANGUAGE COMPILER
*         PASS 2 - READ INTERMEDIATE STUFF, PRODUCE FORTH OUTPUT
*
*****************************************************************************
************************* DATA DEFINITIONS **********************************
*
  DATA('LINK(NEXT, VALUE)');   DATA('VAR_REC(NOCC, TYPE, OFFSET)')
  DATA('FORTH(MATCH, GOAL, OFF)')
*
************************* INITIALIZATIONS ***********************************
*
  b = ' ';  t = " ' ";  c = ' , ';   &ALPHABET  TAB(127) LEN(1) . sp
  IDT = TABLE(70)
   IDT<'feat 2'> = 'FEAT';  IDT<'cons 2'> = 'CONS';  IDT<'nil 0'> = 'NIL';
   IDT<'typatom 1'> = 'TYPATOM';  IDT<'typlex 1'> = 'TYPLEX';
   IDT<'typcr 1'> = 'TYPCR';  IDT<'fail 0'> = 'FEHL';
   IDT<'! 0'> = 'CUT';  IDT<'s 5'> = 'S';  IDT<'np 4'> = 'NP'
   IDT<'qp 2'> = 'QP';  IDT<'nnode 2'> = 'NNODE';  IDT<'pp 3'> = 'PP'
   IDT<'p 1'> = 'P';  IDT<'vg 4'> = 'VG';  IDT<'v 2'> = 'V'
   IDT<'date 3'> = 'DATE';  IDT<'lexeq 2'> = 'LEXEQ'
   IDT<'dp 3'> = 'DP'
  PREDT = TABLE(50);  PREDN = TABLE(50)
   PREDN<'FEAT'> = 'FEAT';  PREDN<'CONS'> = 'CONS';  PREDN<'NIL'> = 'NIL';
   PREDN<'TYPATOM'> = 'TYPATOM';  PREDN<'TYPLEX'> = 'TYPLEX';
   PREDN<'TYPCR'> = 'TYPCR';  PREDN<'FEHL'> = 'FEHL';  PREDN<'CUT'> = 'CUT';
   PREDN<'S'> = 'S';  PREDN<'NP'> = 'NP';  PREDN<'QP'> = 'QP';
   PREDN<'NNODE'> = 'NNODE';  PREDN<'PP'> = 'PP';  PREDN<'P'> = 'P';
   PREDN<'VG'> = 'VG';  PREDN<'V'> = 'V';  PREDN<'DATE'> = 'DATE';
   PREDN<'LEXEQ'> = 'LEXEQ'
   PREDN<'DP'> = 'DP'
  INPUT(. INSTR, 1, 'SY: TEMPLATE. INT')
  OUTPUT(. OUTSTR, 2, 'SY: TEMPLATE. 4TH')
  OUTPUT(. CONSOL, 7)
  &ANCHOR = 1
  PID = 0;  AID = 0;  FID = 0;  CID = 0
  PREDLST =
  LINC = 2; GINC = 4
  SETEXIT('ERR');  &ERRLIMIT = 1;  &STLIMIT = -1
*
************************ FUNCTION DEFINITIONS ********************************
*    DEFINES
  DEFINE('EMITPR(PRED)');   DEFINE('NEW(I)');   DEFINE('S_(NAME)T1,T2,T3,T4')
  DEFINE('PUSH(X)');   DEFINE('POP()');   DEFINE('TOP()')
  DEFINE('TID(LET,NAME,ARITY)');
  DEFINE('EMIT(LINE)L');   DEFINE('DEC(I)');
  DEFINE('FIN_HEAD()')
  :(ENDEF)
*
*    DEFINITIONS
ERR  DUMP(2)   :(ABORT)

FIN_HEAD
  T1 = MATCH(POP());   ARGNO = POP();   T3 = TID('F',POP(),ARGNO);
  T4 = 'C' NEW(.CID);   CLOUT = CLOUT T4 b b;
  PREDT<T3> = PREDT<T3> t T4 " TRY"
  PUSH() = T1;   T1 = 16 + 2 * (NVAR - NGLOB);   T2 = 4 * NGLOB
  CLOUT = CLOUT T2 b T1 " HEAD  " POP()
  CLOUT = GT(T1 - LOFF,0) CLOUT T1 b LOFF " LOCALINIT "
  CLOUT = GT(T2 - GOFF,0) CLOUT T2 b GOFF " INIT "
  :(RETURN)

EMIT
  OUTSTR = LT(SIZE(LINE),81) LINE   :S(RETURN)
  L = 80
EMIT_1
  LINE  TAB(*L) $ DEC(.L) . OUTSTR (' ' | RPOS(0)) =   :F(EMIT_1)
  L = 75
  LINE = GT(SIZE(LINE),75) '      ' LINE   :S(EMIT_1)
  OUTSTR = '      ' LINE :(RETURN)
DEC
  $I = $I - 1;  DEC = .DUMMY   :(NRETURN)

PUSH
  PUSH_POP = LINK(PUSH_POP,X);   PUSH = .VALUE(PUSH_POP)   :(NRETURN)

POP
  IDENT(PUSH_POP)   :S(FRETURN)
  POP = VALUE(PUSH_POP);   PUSH_POP = NEXT(PUSH_POP)   :(RETURN)

TOP
  TOP = DIFFER(PUSH_POP) .VALUE(PUSH_POP)   :F(FRETURN) S(NRETURN)

TID
  DIFFER(TID = IDT<NAME b ARITY>)   :S(RETURN)
  TID = LET NEW(.$(LET 'ID'));   IDT<NAME b ARITY> = TID
  EMIT(ARITY ' FNLIT ' TID b NAME ''');   EQ(ARITY,0)   :S(RETURN)
  PREDN<TID> = 'P' NEW(.PID);   EMIT('RECURS ' PREDN<TID>)
  PREDT<TID> = ':R ' PREDN<TID> '  ENTER  ';  PREDLST = LINK(PREDLST,TID)
  : (RETURN)

NEW
  NEW = $I;  $I = $I + 1  : (RETURN)

EMITPR
  &ANCHOR = 0
  PREDT<$PRED>  'TRY' RPOS(0) = 'TRYLAST ;'  :F(EMITPR_R)
  EMIT(PREDT<$PRED>);
EMITPR_R
  &ANCHOR = 1  : (RETURN)

*
************************* SEMANTIC ROUTINES ***********************************
*   MAIN ROUTINE DEFINITION
S_  : ($('P2_' NAME))

*   SEMANTIC ROUTINES
*
P2_VAR
  T2 = 'v' INSTR;  IDENT(OFFSET($T2))  :F(P2_VAR_REF)
  EQ(NOCC($T2),1)  :S(P2_VDVAR)
  T3 = TYPE($T2) 'OFF'; OFFSET($T2) = $T3; $T3 = $T3 + $(TYPE($T2) 'INC');
  PUSH() = FORTH(OFFSET($T2) b ARGNO " U" TYPE($T2) 'VAR  ',
+            OFFSET($T2) c t (IDENT(TYPE($T2),'L') 'LOCAL', 'VAR') 'ACCESS ,  ',
+            OFFSET($T2))

+ : (RETURN)
P2_VAR_REF
  PUSH() = FORTH(OFFSET($T2) b ARGNO " U" TYPE($T2) 'REF  ',
+            OFFSET($T2) c t (IDENT(TYPE($T2),'L') 'LOCAL', 'VAR') 'ACCESS ,  ')
+ : (RETURN)

P2_VDVAR
  PUSH() = FORTH(, "0 , ' VOIDACCESS ,  ")  : (RETURN)

P2_ATOM
  T1 = INSTR;  T2 = TID('A',T1,0);
  PUSH() = FORTH(t T2 b ARGNO " UATOM  ", t T2 c "' ATOMACCESS ,  ")
  : (RETURN)

P2_INT
  T1 = INSTR;
  PUSH() = FORTH(T1 b ARGNO " UINT  ", T1 c "' INTACCESS ,  ")
  : (RETURN)

P2_CONSI
  ARGNO = 4
  : (RETURN)
```

```
P2_INCL
  EL_CT = EL_CT + 1
  : (RETURN)

P2_NIL
  PUSH() = FORTH("' NIL "  ARGNO " UATOM  ", "' NIL , ' ATOMACCESS ,  ")
  : (RETURN)

P2_LISTI
  NEST = NEST + 1;   EL_CT = 1; PUSH() = ARGNO;   ARGNO = 0
  : (RETURN)

P2_LISTF
  NEST = NEST - 1;   T2 = POP();   T1 = POP();   T3 = 'S' NEW(.SID)
  HI1 = IDENT(HI1) OFF(T2);   LO1 = DIFFER(OFF(T2)) OFF(T2)
  HI1 = IDENT(HI1) OFF(T1);   LO1 = DIFFER(OFF(T1)) OFF(T1)
  EMIT("HERE   " GOAL(T1) GOAL(T2))
  EMIT("HERE  [: " MATCH(T1) MATCH(T2) "RETURN :]")
  EMIT("IARRAY " T3 " ' CONS , , ,")
P2_LISTF_LP
  EQ(EL_CT = EL_CT - 1,0)   :S(P2_LISTF_ND)
  T1 = POP()
  HI1 = IDENT(HI1) OFF(T1);   LO1 = DIFFER(OFF(T1)) OFF(T1)
  EMIT("HERE   " GOAL(T1) t T3 " , ' MOLACCESS ,")
  EMIT("HERE  [: " MATCH(T1) b t T3 " 4 USKEL  RETURN :]")
  T3 = 'S' NEW(.SID);   EMIT("IARRAY " T3 " ' CONS , , ,")  :(P2_LISTF_LP)
P2_LISTF_ND
  ARGNO = POP()
  PUSH() = FORTH(t T3 b ARGNO " USKEL  ", t T3 c "' MOLACCESS ,  ")
  : (RETURN)

P2_INC
  ARGNO = ARGNO + 4
  : (RETURN)

P2_SKELI
  PUSH() = INSTR;   PUSH() = ARGNO;   ARGNO = 0;
  NEST = NEST + 1
  : (RETURN)

P2_SKELF
  NEST = NEST - 1;   T2 = ;   T3 = ;   T4 = (ARGNO / 4) + 1
P2_SKELF_LP
  T1 = POP();   T2 = MATCH(T1) T2;   T3 = GOAL(T1) T3;
  HI1 = IDENT(HI1) OFF(T1);   LO1 = DIFFER(OFF(T1)) OFF(T1)
  GE(ARGNO = ARGNO - 4,0)   :S(P2_SKELF_LP)
  ARGNO = POP();
  GT(NEST,0) :S(P2_SKELF_N)
  PUSH() = T4
  PUSH() = FORTH(T2,T3)   : (RETURN)
```

```
P2_SKELF_N
  T1 = 'S' NEW(.SID);  EMIT('HERE ' T3);  EMIT('HERE  [: ' T2 '  RETURN :]');
  T2 = POP();  T3 = TID('F',T2,T4);
  EMIT("IARRAY " T1 "  ' " T3 " , , ,")
  T3 = (DIFFER(HI1) EQ(NEST,1) (HI1 + 4) b LO1 ' INIT  ', '')
  HI1 = EQ(NEST,1)
  PUSH() = FORTH(T3 t T1 b ARGNO " USKEL  ", t T1 c "' MOLACCESS ,  ")
  :(RETURN)

P2_GOAL
  T1 = GOAL(POP());  T2 = POP();  T3 = TID('F',POP(),T2);
  &ANCHOR = 0
P2_GOAL_RP  T1  'VAR' = 'GLOBAL'  :S(P2_GOAL_RP)
  &ANCHOR = 1
  CLOUT = CLOUT "LIT " PREDN<T3> " CALL [. " T1 " .]  "
  :(RETURN)

P2_CNEK
  FIN_HEAD()
  CLOUT = CLOUT "NECK  "
  :(RETURN)

P2_CFOOT
  CLOUT = CLOUT (ARGNO * 4) " FOOT "
  :(RETURN)

P2_CNC
  FIN_HEAD()
  CLOUT = CLOUT LOFF " NECKCUT  "  :(RETURN)

P2_CNCF
  FIN_HEAD()
  CLOUT = CLOUT (ARGNO * 4) " NECKCUTFOOT  "  :(RETURN)

P2_CNF
  FIN_HEAD()
  CLOUT = CLOUT (ARGNO * 4) ' NECKFOOT '  :(RETURN)

P2_EOCL
  DONE = 'YES'
  EMIT(CLOUT ';')
  POP()  :F(RETURN);  DUMP(2)
  :(RETURN)

P2_INI
  LOFF = 16;  GOFF = 0;  MID = 0;  GID = 0;  SID = 0;
  CLOUT = ': ';  ARGNO = 0;  NEST = 0; HI1 =
  :(RETURN)
*
ENDEF
```

```
*
*******************************************************************************
*                              MAIN PROGRAM                                   *
*******************************************************************************
  OUTSTR = "' RDLN 2- WNEXT ! 2 WMODE !"
  OUTSTR = "SC.V GRMDF"
** READ AND COMPILE A CLAUSE
CMPL
  INSTR  :F(EOIN)
*   READ THE VARIABLE LINES, BUILD VARIABLE TABLE "VART"
  VARL = ;   NVAR = 0;   NGLOB = 0
VAR
  INSTR  BREAK(':') . T1 ':' REM . T3  :F(EOVAR)
  IDENT($(T2 = 'v' T1))  :F(VAR_REF)
  VARL = LINK(VARL,T2)
  $T2 = VAR_REC(1, (IDENT(T3,'G') 'F','L'),)
  :(VAR)
VAR_REF
  NE(NOCC($T2) = NOCC($T2) + 1, 2)  :S(VAR_ETC)
  NVAR = NVAR + 1
  DIFFER(TYPE($T2),'F')  :S(VAR_ETC)
  TYPE($T2) = 'G';   NGLOB = NGLOB + 1  :(VAR)
VAR_ETC
  IDENT(T3,'G')  :F(VAR)
  NGLOB = DIFFER(TYPE($T2),'G') NGLOB + 1  :F(VAR)
  TYPE($T2) = 'G'
  :(VAR)
EOVAR
*  READ THE INSTRUCTION LINES AND CALL THE SEMANTIC ROUTINES
  DONE =
DO_LOOP
  S_(INSTR);   IDENT(DONE)  :S(DO_LOOP)
*  EMPTY CONTENTS OF SYMBOL TABLE
EMPTY
  DIFFER(VARL)  :F(CMPL);   $(VALUE(VARL)) = ; VARL = NEXT(VARL)  :(EMPTY)
** END OF INPUT, OUTPUT PREDICATE CLAUSES
EOIN
  IDENT(PREDLST)  :S(OUT)
  EMITPR(.VALUE(PREDLST));   PREDLST = NEXT(PREDLST)  :(EOIN)
OUT
  OUTSTR = "WMODE OSET SC.I GRMDF"
END
```

6.0 ERL Templates

```
***** Event Representation Templates and auxiliaries as of Jan 17, 1979. *****

| The top-level procedure

do([Tree]):-  build_ER(Tree,ER), type_ER(ER).
```

```
| The 'build_ER' procedure

build_ER  (s(Subj,Vbgr,Obj,Compl,Vmods),temp(Name,ER)):-
                   find_t_name(Vbgr,Name),
                   construct(Name,s(Subj,Vbgr,Obj,Compl,Vmods),ER).

| The 'construct' procedure

construct('DEPLOY',s(Subj,Vbgr,Obj,Compl,Vmods),  [OB1,D1,DTG]):-
         object1 (Subj, OB1),
         destination1 (Vmods,D1),
         construct('DTG',Vmods,DTG).

construct('DTG',List, [TI,DT]):-
         time(List,TI),
         date(List,DT).

construct('ENROUTE',s(Subj,Vbgr,Obj,Compl,Vmods), [OB1,MI,D1,DTG]):-
         object1 (Subj, OB1),
         mission(s(_,_,Obj,Compl,Vmods),MI),
         destination1 (Vmods, D1),
         construct('DTG',Vmods,DTG).

construct('PRECEDE',  s(Subj,_,Obj,_,_), [E1,E2]):-
         build_ER(Subj,E1),
         build_ER(Obj,  E2).

construct('AIRCRAFT',np(Det,L1,Head,L2), [EQ,NA,SUB,SB,SET]):-
         equipment(L1,Head,EQ),
         nationality(L1,'NATION',NA),
         subordination(L2,SUB),
         stagingbase(L2,SB),
         setspec(Det,SET).


construct('DTG',Vmods,[TI,DT]):- time(Vmods, TI), date(Vmods,DT).

| Procedures for filling in template slots

date(Vmods,slot('Date=', [L,W,Day,Month,Year])):-
         member(pp(L,W,date(Day,Month,Year)),Vmods).
date(_, nil).

destination1(Vmods,slot('Destination=',Slot)):- fill_slot(Vmods, ['TO'],'LOC',
                                                                          Slot).
destination2 (X,Y):-  destination1(X,Y).
destination2 (_,nil).

equipment(List,  nnode(W,_),  slot('...Equipment=', [List,W])):-
         feat(W,'ACRAFT').
```

```
nationality(List, Feature,slot('...Nationality=', W)):-
        member(nnode(W,_),List), feat(W,Feature).
nationality(List, Feature,slot('...Nationality=',W)):- member(W,List),
                                                        feat(W,Feature).
nationality(_,_,nil).

object1 (NP, slot('Object:',Slot)):-    test_nhead(NP,'ACRAFT'),
                            construct('AIRCRAFT',NP, Slot).

setspec(dp(_,_,Num),slot('...Number=',Num)).
setspec(_, nil).

stagingbase(List,slot('...Stagingbase=',Slot)):-   fill_slot(List, ['AT'],
                                                   'LOC',Slot).
stagingbase(_,nil).

subordination( List,slot('...Subordination=',Slot)):- fill_slot(List, ['FROM'],
                                                      'SUBNUM',Slot).
subordination(_,nil).

mission(s(_,_,_,_,Vmods),slot('Mission=',Slot)):-
        fill_slot(Vmods, ['AFTER', 'FROM','IN','ON'], 'ACTY', Slot).
mission(_,nil).

time(Vmods,slot('Time=', Slot)):-
        find_time(Vmods,['AT','BETWEEN','BY','DURING','SINCE'],'TYME',Slot).
time(Vmods,slot('Time=',Slot)):-
        find_time(Vmods,['AT','BETWEEN','BY','DURING','SINCE'],'4DIG',Slot).
time(Vmods,slot('Time=',Slot)):-
        fill_slot(Vmods,['AT','BETWEEN','BY','DURING','SINCE'],'TYME',Slot).
time(Vmods,slot('Time=',Slot)):-
        fill_slot(Vmods, 'TYME',Slot).
time(_,nil).

| Other Procedures

fill_slot(List, Preplist, Feature, [L1,Prep,NP]):-
        member (pp(Li,Prep,NP),List),
        member(Prepa, Preplist), lexeq(Prep, Prepa),
        test_nhead(NP, Feature).
fill_slot(List, Feature,W):-
        member(Wa, List), lexeq(W, Wa),
        feat(W,'ADVB'),
        feat(W, Feature).
fill_slot(NP, Feature,NP):-   test_nhead(NP,'LOC').

find_feat(W,L,Y):-   member(Y,L), feat(W,Y).

find_t_name(vg(_,_,_,W),Name):-
        find_feat(W,['ARRIVE','DEPART','DEPLOY','ENROUTE','FLIGHT',
                              'LOCATE','PENETRATE','PRECEDE','RECOVER',
                              'RETURN'],Name).

find_t_name(nnode(W,_),Name):-  find_feat(W, ['AIRCRAFT'],Name).

find_time(List, Preplist, Feature, [L1,W,L2]):-
        member(pp(L1,W,L2),List),
        member(Wa, Preplist), lexeq(W, Wa),
        member(X,L2),
        feat(X,Feature).

member(X, [X,.._]).
member(X, [_,..L]):-  member(X,L).

| Procedure to type event records (type leaves of structure)

type_ER(temp(N,L)):-  typcr(0), typatom('Event:'), typatom(N), type_ER(L).
type_ER(slot(S,L)):-  typcr(0), typatom(S), type_ER(L).
type_ER([X,..L]):-  type_ER(X), type_ER(L).
type_ER(nil).
type_ER(X):-  typlex(X).
type_ER(X):-  typatom(X).
type_ER(s(A,B,C,D)):-  type_ER(A), type_ER(B), type_ER(C), type_ER(D).
type_ER(np(A,B,C,D)):-  type_ER(A), type_ER(B), type_ER(C), type_ER(D).
type_ER(qp(A,B)):-  type_ER(A), type_ER(B).
type_ER(dp(A,B,C):- type_ER(A), type_ER(B), type_ER(C).
type_ER(nnode(A,B)):-  type_ER(A), type_ER(B).
type_ER(v(A,B)):-  type_ER(A), type_ER(B).
type_ER(pp(A,B,C)):-  type_ER(A), type_ER(B), type_ER(C).
type_ER(p(A)):-  type_ER(A).
type_ER(vg(A,B,C,D)):-  type_ER(A), type_ER(B), type_ER(C), type_ER(D).
type_ER(date(A,B,C)):-  type_ER(A), type_ER(B), type_ER(C).

test_nhead(np(_,_,nnode(W,_),_),Feature):- feat(W,Feature).
```

Appendix B - Programming in Logic with the Prolog Language

1.0 Introduction

The following paragraphs are extracted from a provisional version of the *User's Guide to DECsystem-10 PROLOG*† and provide an introduction to the syntax and semantics of a certain subset of logic ("definite clauses", also known as "Horn clauses"), and indicates how this subset forms the basis of Prolog.

2.0 Syntax, Terminology and Informal Semantics

2.1 Terms

The data objects of the language are called *terms*. A term is either a *constant*, a *variable* or a *compound* term.

The constants include *integers* such as:-

```
0  1  999  -512
```

In DECsystem-10 Prolog, integers are restricted to the range -2^{17} to $2^{17}-1$, i.e. -131072 to 131071. Besides the usual decimal, or base 10, notation, integers may also be written in any base from 2 to 9, of which base 2 (binary) and base 8 (octal) are probably the most useful. As an example of the notation used:-

```
15  2'1111  8'17
```

all represent the integer fifteen.

Constants also include *atoms* such as:-

```
a  void  =  :=  'Algol-68'  []
```

The symbol for an atom can be any sequence of characters, which should be written in quotes if there is possibility of confusion with other symbols (such as variables, integers). As in conventional programming languages, constants are definite elementary objects, and correspond to proper nouns in natural language.

Variables are distinguished by an initial capital letter or by the initial character "_", eg.

```
X  Value  A  A1  _3  _RESULT
```

If a variable is only referred to once, it does not need to be named and may be written as an "anonymous" variable indicated by a single underline character.

A variable should be thought of as standing for some definite but unidentified object. This is analogous to the use of a pronoun in natural language. Note that a variable is not simply a writeable storage location as in most programming languages; rather it is a local name for some data object, cf. the variable of pure Lisp and identity declarations in Algol-68.

The structured data objects of the language are the compound terms. A compound term comprises a *functor* (called the *principal* functor of the term) and a sequence of one or

---

† This version was prepared by Luis Moniz Pereira of Divisao de Informatica Laboratorio National de Engenharia Civil, Lisbon, and Fernando C.N. Pereira, and David H. D. Warren, both at the Department of Artificial Intelligence, University of Edinburgh, England, and issued April 1978.

more terms called *arguments*. A functor is characterized by its *name*, which is an atom, and its *arity* or number of arguments. For example, the compound term whose functor is named 'point' of arity 3, with arguments *X*, *Y* and *Z*, is written:-

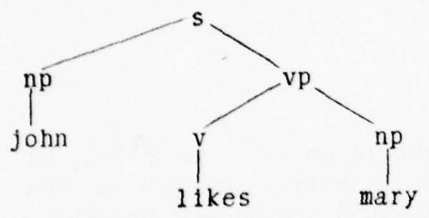
point (*X*,*Y*,*Z*)

Note that an atom is considered to be a functor of arity 0.

Functors are generally analogous to common nouns in natural language. One may think of a functor as a record type and the arguments of a compound term as the fields of a record. Compound terms are usefully pictured as trees. For example, the term:-

s(np(john),vp(v(likes),np(mary)))

would be pictured as the structure:-



Sometimes it is convenient to write certain functors as *operators* - 2-ary functors may be declared as *infix* operators and 1-ary functors as *prefix* or *postfix* operators. Thus it is possible to write, eg.

X+*Y*   (*P*;*Q*)   *X*<*Y*   +*X*   *P*;

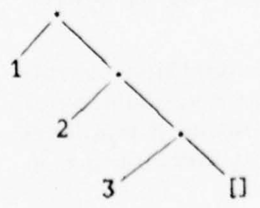
as optional alternatives to:-

+(*X*,*Y*)   ;(*P*,*Q*)   <(*X*,*Y*)   +(*X*)   ;(*P*)

An important class of data structures are the *lists*. These are essentially the same as the lists of Lisp. A list either is the atom:-

[]

representing the empty list, or is a compound term with functor '.' and two arguments which are respectively the head and tail of the list. Thus a list of the first three natural numbers is the structure:-



which could be written, using the standard syntax, as:-

.(1,.(2,.(3,[])))

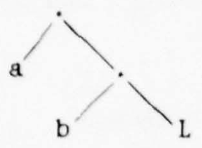
but which is normally written, in a special list notation, as:-

[1,2,3]

The special list notation in the case when the tail of a list is a variable is exemplified by:-

[a,b,..L]

representing:-



For convenience, a further notational variant is allowed for lists of integers which correspond to ASCII character codes. Lists written in this notation are called *strings*. An example is:-

"DECsystem-10"

which represents exactly the same list as:-

[68,69,67,115,121,115,116,101,109,45,49,48]

2.2 Programs

A fundamental unit of a logic program is the *goal* or *procedure call*. Examples are:-

gives(tom,apple,teacher)   reverse([1,2,3],L)   *X<Y*

A goal is merely a special kind of term, distinguished only by the context in which it appears in the program. The (principal) functor of a goal is called a *predicate*. It corresponds roughly to a verb in natural language, or to a procedure name in a conventional programming language.

A logic *program* consists simply of a sequence of statements called *sentences*, which are analogous to sentences of natural language. A sentence comprises a *head* and a *body*. The head either consists of a single goal or is empty. The body consists of a sequence of zero or more goals (i.e., it too may be empty. The body consists of a sequence of zero or more goals (i.e., it too may be empty). If the head is not empty, the sentence is called a *clause*.

If the body of a clause is non-empty, the clause is called a *non-unit clause*, and is written in the form:-

P :- Q, R, S.

where *P* is the head goal and *Q*, *R* and *S* are the goals which make up the body. We can read such a clause either *declaratively* as:-

"*P* is true if *Q* and *R* and *S* are true."

or *procedurally* as:-

"To satisfy goal *P*, satisfy goals *Q*, *R* and *S*."

If the body of a clause is empty, the clause is called a unit clause, and is written in the form:-

P.

where *P* is the head goal. We interpret this declaratively as:-

"*P* is true."

and procedurally as:-

"Goal *P* is satisfied."

A sentence with an empty head is called a *directive*, of which the most important kind is called a *question* and is written in the form:-

?- *P*, *Q*.

where *P* and *Q* are the goals of the body. Such a question is read declaratively as:-

"Are *P* and *Q* true?"

and procedurally as:-

"Satisfy goals *P* and *Q*."

Sentences generally contain variables. Note that variables in different sentences are completely independent, even if they have the same name — i.e., the "lexical scope" of a variable is limited to a single sentence. Each distinct variable in a sentence should be interpreted as standing for an arbitrary entity, or value. To illustrate this, here are some examples of sentences containing variables, with possible declarative and procedural readings:-

(1) employed(X) :- employs(Y,X).

"Any X is employed if any Y employs X."

"To find whether a person X is employed,
find whether any Y employs X."

(2) derivative(X,X,1).

"For any X, the derivative of X with respect to X is 1."

"The goal of finding a derivative for the expression X with
respect to X itself is satisfied by the result 1."

(3) ?- ungulate(X), aquatic(X).

"Is it true, for any X, that X is an ungulate and X is
aquatic?"

"Find an X which is both an ungulate and aquatic."

In any program, the *procedure* for a particular predicate is the sequence of clauses in the program whose head goals have that predicate as principal functor. For example, the procedure for a ternary predicate

```
concatenate([X,..L1],L2,[X,..L3]) :- concatenate (L1, L2, L3).
concatenate ([],L,L).
```

where 'concatenate(L1,L2,L3)' means "the list L1 concatenated with the list L2 is the list L3".

Certain predicates are predefined by *built-in procedures* supplied by the Prolog system. Such predicates are called *evaluable predicates*.

As we have seen, the goals in the body of a sentence are linked by the operator ',' which can be interpreted as conjunction ("and"). It is sometimes convenient to use an additional operator ';', standing for disjunction ("or"). (The precedence of ';' is such that it dominates ' but is dominated by ':-'). An example is the clause:-

```
grandfather(X,Z) :-
    (mother(X,Y); father(X,Y)), father(Y,Z).
```

which can be read as:-

"For any X, Y and Z,
    X has Z as a grandfather if
    either the mother of X is Y or the father of X is Y,
    and the father of Y is Z.

Such uses of disjunction can always be eliminated by defining an extra predicate — for instance the previous example is equivalent to:-

```
grandfather(X,Z) :- parent(X,Y), father (Y,Z).
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
```

— and so disjunction will not be mentioned further in the following, more formal, description of the semantics of clauses.

3.0 Declarative And Procedural Semantics

The semantics of definite clauses should be fairly clear from the informal interpretations already given. However, it is useful to have a precise definition. The *declarative semantics* of definite clauses tells us which goals can be considered true according to a given program, and is defined recursively as follows.

> A goal is *true* if it is the head of some clause instance and each of the goals (if any) in the body of that clause instance is true, where an *instance* of a clause (or term) is obtained by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable.

For example, if a program contains the preceding procedure for 'concatenate', then the declarative semantics tells us that:-

```
concatenate([a],[b],[a,b])
```

is true, because this goal is the head of a certain instance of the first clause for 'concatenate', namely,

```
concatenate([a],[b],[a,b]) :- concatenate([],[b],[b]).
```

and we know that the only goal in the body of this clause instance is true, since it is an instance of the unit clause which is the second clause for 'concatenate'.

Note that the declarative semantics makes no reference to the sequencing of goals within the body of a clause, nor to the sequencing of clauses within a program. This sequencing information is, however, very relevant for the *procedural semantics* which Prolog gives to definite clauses. The procedural semantics defines exactly how the Prolog system will execute a goal, and the sequencing information is the means by which the Prolog programmer directs the system to execute his program in a sensible way. The effect of executing a goal is to enumerate, one by one, its true instances. Here then is an informal definition of the procedural semantics.

> To *execute* a goal, the system searches for the first clause whose head *matches* or *unifies* with the goal. The *unification* process [Robinson 1965] finds the most generel common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then *activated* by executing in turn, from left to right, each of the goals (if any) in its body. If at any time the system fails to find a match for a goal, it *backtracks*, i.e., it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal.

For example, if we execute the goal expressed by the question:-

```
?- concatenate(X,Y,[a,b]).
```

we find that it matches the head of the first clause for 'concatenate', with X instantiated to [a,..X1). The new variable X1 is constrained by the new goal produced, which is the recursive procedure call:-

```
concatenate(X1,Y,[b])
```

Again this goal matches the first clause, instantiating X1 to [b,..X2], and yielding the new goal:-

```
concatenate(X2,Y,[])
```

Now this goal will only match the second clause, instantiating both X2 and Y to []. Since there are no further goals to be executed, we have a solution:-

```
X = [a,b]
Y = []
```

i.e., a true instance of the original goal is:-

```
concatenate([a,b],[],[a,b])
```

If this solution is rejected, backtracking will generate the further solutions:-

```
X = [a]
Y = [b]

X = []
Y = [a,b]
```

in that order, by re-matching, against the second clause for 'concatenate', goals already solved once using the first clause.

3.1 Prolog Control Mechanisms

As is evident from the above, Prolog provides a remarkably simple form of control, which suffices for many practical applications of logic programming. This point was first realized at Marseille and is the basis of the programming language Prolog developed there.

If we think back to the declarative semantics of clauses, it is clear that the order of the goals in a clause and the order of the clauses themselves are both irrelevant to the declarative interpretation. However, these orderings are generally significant in Prolog, as they constitute the main control information.

When the Prolog system is executing a procedure call, the clause ordering determines the order in which the different entry points of the procedure are tried.The goal ordering fixes the order in which the procedure calls in a clause are executed. The "productive" effect of a Prolog computation arises from the process of "matching" a procedure call against a procedure entry point.

Appendix C - MATRES II Lexicon

60

1.0 Lexicon

```
LEXICON
FEATURE 1DIG
FEATURE 2DIG
FEATURE 4DIG
FEATURE ACRAFT
FEATURE ADVB
FEATURE ADJ
FEATURE NOMZ
FEATURE ACTVTY
FEATURE ALT
FEATURE ARRIVE
FEATURE BE
FEATURE BEFORE
FEATURE COMMUNICATION
FEATURE CONFIRM
FEATURE CONTINUE
FEATURE CONJ
FEATURE COPULA
FEATURE DEPART
FEATURE DEPLOY
FEATURE DIR
FEATURE ART
FEATURE EOS
FEATURE EVAL
FEATURE EMOD
FEATURE ENROUTE
FEATURE FLIGHT
FEATURE GO
FEATURE HEAD
FEATURE HAVE
FEATURE IMPACT
FEATURE INTRANS
FEATURE LOC
FEATURE LOCATE
FEATURE LAND
FEATURE LAUNCH
FEATURE MISSILE
FEATURE MO
FEATURE MODAL
FEATURE NUM
FEATURE NUMMOD
FEATURE NATION
FEATURE NATO
FEATURE N
FEATURE OBSERVE
FEATURE ORD
FEATURE PASTP
FEATURE PL
FEATURE POSPRO
FEATURE PRESP
FEATURE PRO
FEATURE PRTCL
FEATURE PREP
FEATURE PENETRATE
FEATURE QUANT
FEATURE RE-ENTER
FEATURE REF
FEATURE RETURN
FEATURE RELPRO
FEATURE SATELLITE
FEATURE SCONJ
FEATURE SCRAFT
FEATURE SERVICE
FEATURE SG
FEATURE SUBNUM
FEATURE SUPER
FEATURE STAGE
FEATURE THATCOMP
FEATURE TJPE
FEATURE TYME
FEATURE TENSED
FEATURE TOCOMP
FEATURE TRANS
FEATURE UNIT
FEATURE VB
FEATURE VPASSIVE
:: ( AIR FORCE) [ N ] .;
:: ( AIR REGIMENT) [ N ] .;
:: ( AIR SPACE) [ N LOC ] .;
:: ( AL JAGHBUB) [ N LOC ] .;
:: ( ALL OF) [ QUANT ] .;
:: ( A MINIMUM OF) [ NUMMOD ] .;
:: ( ARABIAN SEA) [ N LOC ] .;
:: ( AS FAR) [ PREP EMOD ] .;
:: ( AS MANY AS) [ NUMMOD ] .;
:: ( AT LEAST) [ NUMMOD ] .;
:: ( AT MOST) [ NUMMOD ] .;
:: ( BARENTS SEA) [ N LOC ] .;
:: ( BOMBER CORPS) [ N ] .;
:: ( BUFF A) [ N NATO ACRAFT ] .;
:: ( BUFF B) [ N NATO ACRAFT ] .;
:: ( BUFF C) [ N NATO ACRAFT ] .;
:: ( BUFF D) [ N NATO ACRAFT ] .;
:: ( BUFF O) [ N NATO ACRAFT ] .;
:: ( CAPE VERDE ISLANDS) [ N LOC ] .;
:: ( COMMAND AND CONTROL) [ ADJ ] .;
:: ( CONTROL AND REPORTING) [ ADJ ] .;
:: ( EAST OF) [ PREP ] .;
:: ( GULF OF ADEN) [ N LOC ] .;
:: ( GULF OF AQABA) [ N LOC ] .;
:: ( HEAVY BOMBERS) [ N PL ACRAFT ] .;
:: ( HEAVY BOMBER) [ N SG ACRAFT ] .;
:: ( IN CONJUNCTION WITH) [ PREP ] .;
:: ( IN CONNECTION WITH) [ PREP ] .;
:: ( IN REACTION TO) [ PREP ] .;
:: ( INDIAN OCEAN) [ N LOC ] .;
:: ( LACCADIVE ISLANDS) [ N LOC ] .;
:: ( MALDIVE ISLANDS) [ N LOC ] .;
:: ( MANY OF) [ QUANT ] .;
:: ( MEDIUM BOMBER) [ N SG ACRAFT ] .;
:: ( MEDIUM BOMBERS) [ N PL ACRAFT ] .;
:: ( MIRAGE III) [ N TJPE ACRAFT ] .;
:: ( MOST OF) [ QUANT ] .;
:: ( NATIONAL GUARDS) [ N ] .;
:: ( NATIONAL GUARD) [ N ] .;
:: ( NONE OF) [ QUANT ] .;
:: ( NORTH OF) [ PREP ] .;
:: ( NORTHEAST OF) [ PREP ] .;
:: ( NORTHWEST OF) [ PREP ] .;
:: ( RED SEA) [ N LOC ] .;
:: ( SAUDI ARABIAN) [ ADJ NATION ] .;
:: ( SEA OF CRISES) [ N LOC ] .;
:: ( SEYCHELLE ISLANDS) [ N LOC ] .;
:: ( SEYCHELLE ISLAND CHAIN) [ N LOC ] .;
:: ( SMALL SCALE) [ ADJ ] .;
:: ( SOME OF) [ QUANT ] .;
:: ( SOUTH AFRICAN) [ ADJ NATION ] .;
:: ( SOUTH OF) [ PREP ] .;
:: ( SOUTHEAST OF) [ PREP ] .;
:: ( SOUTHWEST OF) [ PREP ] .;
:: ( SOVIET UNION) [ N LOC NATION ] .;
:: ( ST HELENA) [ N LOC ] .;
:: ( TEST RANGE) [ N LOC ] .;
:: ( WEST OF) [ PREP ] .;
:: ( WHITE SEA) [ N LOC ] .;
:: A [ ART ] .;
:: A313 [ N SUBNUM ] .;
:: AA [ ADJ ] .;
:: ABOUT [ ADVB EVAL ] .;
:: ACFT [ N ACRAFT ] .;
:: ACTIVE [ ADJ ACTVTY ] .;
:: ACTIVITY [ N NOMZ ] .;
:: ACTY [ N NOMZ ] .;
:: ADDITIONAL [ ADJ REF ] .;
:: ADX [ N NOMZ ] .;
:: ADZ [ N LOC ] .;
:: AFRICAN [ ADJ ] .;
:: AFTER [ PREP TYME SCONJ ] .;
:: AGAINST [ PREP EMOD ] .;
:: AIR-TO-SURFACE [ ADJ ] .;
:: AIR [ N ] .;
:: AIRBORNE [ ADJ FLIGHT ] .;
:: AIRCRAFT [ N ACRAFT ] .;
:: ALEXANDRIA [ N LOC ] .;
:: ALL [ QUANT ] .;
:: ALONG [ PREP EMOD ] .;
:: ALTITUDE [ N SG ALT ] .;
:: ALTITUDES [ N PL ALT ] .;
:: AN [ ART ] .;
:: AND [ CONJ ] .;
:: APPROXIMATELY [ ADVB EVAL ] .;
:: APR [ N TYME MO ] .;
:: APRIL [ N TYME MO ] .;
:: ARE [ BE COPULA TENSED ] .;
:: AREA [ N SG LOC ] .;
:: AREAS [ N PL LOC ] .;
:: ARRIVED [ VB PASTP ARRIVE ] [ VB TENSED ARRIVE ] .;
:: AS [ PRTCL ] .;
:: ASSOCIATED [ ADJ ] .;
:: ASM [ ADJ ] .;
:: ASW [ ADJ ] .;
:: AT [ PREP EMOD TYME ] .;
:: ATLANTIC [ N LOC ] .;
:: AUG [ N TYME MO ] .;
:: AUGUST [ N TYME MO ] .;
:: AUXILIARY [ ADJ ] .;
:: AVIATION [ N NOMZ ] .;
:: A-4 [ N TJPE ACRAFT ] .;
:: B-75 [ N TJPE ACRAFT ] .;
:: B-75S [ N TJPE ACRAFT ] .;
:: B-60 [ N TJPE ACRAFT ] .;
:: B-60'S [ N TJPE ACRAFT ] .;
:: B-61 [ N TJPE ACRAFT ] .;
:: B-63 [ N TJPE ACRAFT ] .;
:: B-63'S [ N TJPE ACRAFT ] .;
:: B-63S [ N TJPE ACRAFT ] .;
:: B-67S [ N TJPE ACRAFT  ] .;
:: B-80 [ N TJPE ACRAFT ] .;
:: B-TYPE [ N TJPE ACRAFT ] .;
:: B-TYPES [ N TJPE ACRAFT ] .;
:: BACTERIOLOGICAL [ ADJ ] .;
:: BAIKONUR [ N LOC ] .;
:: BARFLY [ N NATO ACRAFT ] .;
:: BASE [ N LOC ] .;
:: BC254 [ N SUBNUM ] .;
:: BE [ BE COPULA ] .;
:: BEACON [ N NATO ACRAFT ] .;
:: BEAGLE [ N NATO ACRAFT ] .;
:: BED [ N ] .;
:: BEEN [ BE COPULA PASTP ] .;
:: BEETLES [ N ] .;
:: BEFORE [ PREP TYME SCONJ ] .;
:: BEING [ BE COPULA PRESP ] .;
:: BETWEEN [ PREP EMOD TYME ] .;
:: BOMBER [ N SG ACRAFT ] .;
:: BOMBERS [ N PL ACRAFT ] .;
:: BORDER [ N LOC ] .;
:: BOUNDED [ ADJ ] .;
:: BUFF [ N NATO ACRAFT ] .;
:: BUJUMBURA [ N LOC ] .;
:: BUTTER [ N NATO ACRAFT ] .;
:: BY [ PREP EMOD TYME ] .;
:: CAPETOWN-BASED [ ADJ ] .;
:: CAPETOWN [ N LOC ] .;
:: CAPSULE [ N ] .;
:: CENTER [ N LOC ] .;
:: CENTRAL [ ADJ ] .;
:: COAST [ N LOC ] .;
:: COLLECTION [ N NOMZ ] .;
:: COMBAT [ N NOMZ ] .;
:: COMBATANT [ N ] .;
:: COMBATANTS [ N PL ] .;
:: COMMUNICATION [ N ] .;
:: COMPLEX [ N LOC ] .;
:: CONDUCTING [ VB TRANS PRESP FLIGHT ] .;
:: CONDUCTED [ VB TRANS PASTP FLIGHT ] [ VB TRANS TENSED FLIGHT ] .;
:: CONFIRMED [ VB TRANS PASTP CONFIRM ] [ VB TRANS TENSED CONFIRM ] .;
:: CONGO [ N LOC ] .;
:: CONTAINING [ VB TRANS PRESP ] .;
:: CONTINUED [ VB TRANS PASTP ] [ VB TRANS TENSED ] .;
:: CONTINUING [ VB TRANS PRESP DIR CONTINUE ] .;
:: CONTINENTAL [ ADJ ] .;
:: CONTROLLER [ N ] .;
:: CORNER [ N ] .;
:: COSMODROME [ N LOC ] .;
:: COSMOS-605 [ N SATELLITE ] .;
:: COSMOS-629 [ N SATELLITE ] .;
:: COSMOS-706 [ N SATELLITE ] .;
:: COSMOS-722 [ N SATELLITE ] .;
:: CRAFT [ N SCRAFT ] .;
:: CURRENTLY [ ADVB REF TYME ] .;
:: DAMASCUS [ N LOC ] .;
:: DEC [ N TYME MO ] .;
:: DECEMBER [ N TYME MO ] .;
:: DEFENSIVE [ ADJ ] .;
:: DELTA-CLASS [ ADJ ] .;
:: DEPARTED [ VB TRANS PASTP DEPART ] [ VB TRANS TENSED DEPART ] .;
:: DEPLOYED [ VB TRANS PASTP DEPLOY ] [ VB TRANS TENSED DEPLOY ] .;
:: DEPLOYMENTS [ N PL NOMZ ] .;
:: DESTINATION [ N ] .;
:: DIVISION [ N ] .;
:: DJIBOUTI [ N LOC ] .;
:: DOWNED [ VB TRANS PASTP ] [ VB TRANS TENSED ] .;
:: DURING [ PREP EMOD TYME ] .;
:: E1651D [ N SUBNUM ] .;
:: EABC [ N NATION ] .;
:: EAFAF [ N NATION SERVICE ] .;
:: EARLIER [ ADVB TYME REF ] .;
:: EARLY [ ADVB TYME ] [ ADJ TYME ] .;
:: EARTH [ N ] .;
:: EAST [  ADVB DIR ] [ ADJ ] .;
:: EASTERN [ ADJ ] .;
:: EGYPTIAN [ ADJ NATION ] .;
:: EIGHT [ NUM ] .;
:: ENROUTE [ ADJ TOCOMP ENROUTE ] .;
:: ENTEBBE [ N LOC ] .;
:: EQUIPMENT [ N ] .;
:: ETBF [ N LOC ] .;
:: ETHIOPIAN [ ADJ ] .;
:: EXERCISE [ N NOMZ ] .;
:: F-4 [ N TJPE ACRAFT ] .;
:: F-4E [ N TJPE ACRAFT ] .;
:: F-5E [ N TJPE ACRAFT ] .;
:: FEB [ N TYME MO ] .;
:: FEBRUARY [ N TYME MO ] .;
:: FERRY [ N ] .;
:: FIGHTER-BOMBERS [ N PL ACRAFT ] .;
:: FIGHTER [ N SG ACRAFT ] .;
:: FIGHTERS [ N PL ACRAFT ] .;
:: FIRED [ VB TRANS PASTP LAUNCH ] [ VB TRANS TENSED LAUNCH ] .;
:: FLEET [ N ] .;
:: FLEW [ VB TRANS TENSED DIR FLIGHT ] .;
:: FLIGHT [ N SG NOMZ ] .;
:: FLIGHTS [ N PL NOMZ ] .;
:: FLOGGER [ N NATO ACRAFT ] .;
:: FODDER [ N NATO ACRAFT ] .;
:: FOLLOWING [ SCONJ ] .;
:: FOR [ PREP ] .;
:: FOUR [ NUM ] .;
:: FRESCO [ N NATO ACRAFT ] .;
:: FROM [ PREP EMOD TYME ] .;
:: GENERAL [ ADJ ] .;
:: GROUP [ N ] .;
:: GULU-BASED [ ADJ ] .;
:: GULU [ N LOC ] .;
:: HAD [ HAVE TENSED ] [ VB TRANS PASTP ] [ VB TRANS TENSED ] .;
:: HAIFA [ N LOC ] .;
:: HAS [ HAVE TENSED ] [ VB TRANS TENSED ] .;
:: HAVE [ HAVE ] [ HAVE TENSED ] [ VB TRANS ] [ VB TRANS TENSED ] .;
:: HAVING [ VB TRANS PRESP ] .;
:: HEADING [ VB PRESP DIR HEAD ] .;
:: HERMETICALLY [ ADVB ] .;
:: HOMEBASE [ N LOC ] .;
:: HOUR [ N TYME UNIT ] .;
:: HOURS [ N TYME UNIT ] .;
:: HR [ N TYME UNIT ] .;
:: IL-28 [ N TJPE ACRAFT ] .;
:: IN [ PREP EMOD ] .;
:: INDEPENDENT [ ADJ ] .;
:: INFORMED [ VB TRANS PASTP THATCOMP COMMUNICATION ]
            [ VB TRANS TENSED THATCOMP COMMUNICATION ] .;
:: INTELLIGENCE [ N NOMZ ] .;
:: INTO [ PREP EMOD ] .;
:: INVOLVING [ VB TRANS PRESP ] .;
:: IRANIAN [ ADJ NATION ] .;
:: IS [ BE COPULA TENSED ] .;
:: ISRAEL [ N LOC NATION ] .;
:: ISRAELI [ ADJ NATION ] .;
:: JAN [ N TYME MO ] .;
:: JANUARY [ N TYME MO ] .;
:: JUBA [ N LOC ] .;
:: JUN [ N TYME MO ] .;
:: JUNE [ N TYME MO ] .;
:: JUL [ N TYME MO ] .;
:: JULY [ N TYME MO ] .;
:: JUST [ ADVB ] .;
:: KATHMANDU [ N LOC ] .;
:: KB252 [ N SUBNUM ] .;
:: KE843 [ N SUBNUM ] .;
:: KENYA [ N LOC NATION ] .;
:: KENYAN [ ADJ NATION ] .;
:: KFIR [ N TJPE ACRAFT ] .;
:: KILOMETERS [ N PL UNIT ] .;
:: KINSHASA [ N LOC ] .;
:: KM [ N SG UNIT ] .;
:: KMS [ N PL UNIT ] .;
:: LACCADIVES [ N LOC ] .;
:: LANDED [ VB PASTP LAND ] [ VB TENSED LAND ] .;
:: LANDMASS [ N LOC ] .;
:: LAST [ ADJ ] .;
:: LATE [ ADJ TYME ] .;
:: LAUNCHED [ VB TRANS PASTP LAUNCH ] [ VB TRANS TENSED LAUNCH ] .;
:: LEBANON [ N LOC NATION ] .;
:: LIBYAN [ ADJ NATION ] .;
:: LIVING [ ADJ ] .;
:: LOCATED [ ADJ VB TRANS PASTP LOCATE ] [ VB TRANS TENSED LOCATE ] .;
:: LUNA-23 [ N SATELLITE ] .;
:: LUNAR [ ADJ ] .;
:: MALAGASY [ N LOC ] .;
:: MANEUVERABLE [ ADJ ] .;
:: MANNED [ ADJ ] .;
:: MANY [ QUANT ] .;
:: MAR [ N TYME MO ] .;
:: MARCH [ N TYME MO ] .;
:: MARITIME [ ADJ ] .;
:: MASSAWA [ N LOC ] .;
:: MAURITIUS [ N LOC ] .;
:: MAY [ MODAL ] [ N TYME MO ] .;
:: METERS [ N PL UNIT ] .;
:: MID [ ADJ ] .;
:: MIG-17 [ N TJPE ACRAFT ] .;
:: MIG-21 [ N TJPE ACRAFT ] .;
:: MIG-23 [ N TJPE ACRAFT ] .;
:: MILES [ N UNIT ] .;
:: MILITARY [ ADJ ] .;
:: MISSILE [ N MISSILE ] .;
:: MISSION [ N NOMZ ] .;
:: ML-28 [ N  TJPE ACRAFT ] .;
:: MOGADISHU [ N LOC ] .;
:: MOMBASA-BASED [ ADJ ] .;
:: MOMBASA [ N LOC ] .;
:: MONTH [ N TYME ] .;
:: MORNING [ N TYME ] .;
:: MOST [ QUANT ] .;
:: MUSHROOM [ N ] .;
:: NAIROBI-BASED [ ADJ ] .;
:: NAIROBI [ N LOC ] .;
:: NATIONAL [ ADJ ] .;
:: NATURE [ N ] .;
:: NAUTICAL [ ADJ ] .;
:: NAVIGATIONAL [ ADJ ] .;
:: NEAR [ PREP EMOB ] .;
:: NEPAL [ N LOC NATION ] .;
:: NIGERIAN [ ADJ NATION ] .;
:: NINE [ NUM ] .;
:: NM [ N SG UNIT ] .;
:: NMS [ N PL UNIT ] .;
:: NO [ QUANT ] .;
:: NORMAL [ ADJ ] .;
:: NORTH [ ADVB DIR ] [ ADJ ] .;
:: NORTHEAST [ ADVB DIR ] [ ADJ ] .;
:: NORTHERN [ ADJ ] .;
:: NORTHWEST [ ADVB DIR ] [ ADJ ] .;
:: NORTHWESTERN [ ADJ ] .;
:: NOTED [ VB TRANS PASTP OBSERVE ] [ VB TRANS TENSED OBSERVE ] .;
:: NOV [ N TYME MO ] .;
:: NOVEMBER [ N TYME MO ] .;
:: NYANJA [ N LOC ] .;
:: OCEAN [ N LOC ] .;
:: OCT [ N TYME MO ] .;
:: OCTOBER [ N TYME MO ] .;
:: OF [ PREP ] .;
:: ON [ PREP EMOD ] .;
:: ONE [ NUM ] .;
:: OPEN [ ADJ ] .;
:: OPERATED [ VB PASTP ACTVTY ] [ VB TENSED ACTVTY ] .;
:: OPERATING [ VB PRESP ACTVTY ] .;
:: OPERATIONS [ N NOMZ ] .;
:: ORBIT [ N LOC ] .;
:: OVER [ PREP EMOD ] .;
:: PENETRATED [ VB TRANS PASTP PENETRATE ] [ VB TRANS TENSED PENETRATE ] .;
:: PENETRATION [ N NOMZ ] .;
:: PERFORMING [ VB TRANS PRESP ACTVTY ] .;
:: PERIOD [ N TYME ] .;
:: PHANTOM [ N NATO ACRAFT ] .;
:: PILOTS [ N PL ] .;
:: PLESETSK [ N LOC ] .;
:: POSSIBLE [ ADJ EVAL ] .;
:: POSSIBLY [ ADVB EVAL ] .;
:: PRECEDED [ VB TRANS PASTP BEFORE ] [ VB TRANS TENSED BEFORE ] .;
:: PRECEDES [ VB TRANS TENSED BEFORE ] .;
:: PRESENTLY [ ADVB REF TYME ] .;
:: PRETORIA-BASED [ ADJ ] .;
:: PRETORIA [ N LOC ] .;
:: PREVIOUS [ ADJ REF TYME ] .;
:: PREVIOUSLY [ ADVB REF TYME ] .;
:: PRIMARILY [ ADVB EVAL ] .;
:: PROBABLE [ ADJ EVAL ] .;
:: PROBABLY [ ADVB EVAL ] .;
:: PROCEEDING [ VB PRESP DIR FLIGHT ] .;
:: RATS [ N PL ] .;
:: RECONNAISSANCE [ N NOMZ ] .;
:: RECOVERABLE [ ADJ ] .;
:: RE-ENTER [ VB TRANS TENSED RE-ENTER ] .;
:: REGT [ N SG ] .;
:: REGIMENT [ N SG ] .;
:: REGIMENTS [ N PL ] .;
:: REMAIN [ COPULA TENSED ] .;
:: REMAINED [ COPULA TENSED ] .;
:: REPORTING [ VB TRANS PRESP ] .;
:: REPRESENTING [ VB TRANS PRESP ] .;
:: RETURNED [ VB PASTP RETURN ] [ VB TENSED RETURN ] .;
:: RETURNING [ VB PRESP RETURN ] .;
:: RGT [ N SG ] .;
:: RIYADH [ N LOC ] .;
:: ROUTINELY [ ADVB ] .;
:: S1234B [ N SUBNUM ] .;
:: SA554 [ N SUBNUM ] .;
:: SA622 [ N SUBNUM ] .;
:: SAFAF [ N NATION SERVICE ] .;
:: SAFLT [ N NATION SERVICE ] .;
:: SATELLITE [ N SATELLITE ] .;
:: SAM-3 [ N MISSILE ] .;
:: SAME [ ADJ ] .;
:: SC462 [ N SUBNUM ] .;
:: SCRAMBLED [ ADJ ] .;
:: SEALED [ ADJ ] .;
:: SEP [ N TYME MO ] .;
:: SEPTEMBER [ N TYME MO ] .;
:: SEYCHELLES [ N LOC ] .;
:: SIBERIA [ N LOC ] .;
:: SIMULATED [ ADJ ] .;
:: SINCE [ PREP EMOD TYME ] .;
:: SIWAH [ N LOC ] .;
:: SIX [ NUM ] .;
:: SKYHAWK [ N NATO ACRAFT ] .;
:: SOFTLANDED [ VB TRANS PASTP LAND ] [ VB TRANS TENSED LAND ] .;
:: SOMALIA [ N LOC NATION ] .;
:: SOME [ QUANT ] .;
:: SOUTH [ ADVB DIR ] [ ADJ ] .;
:: SOUTHERN [ ADJ ] .;
:: SOUTHEAST [ ADVB DIR ] [ ADJ ] .;
:: SOUTHWEST [ ADVB DIR ] [ ADJ ] .;
:: SOUTWESTERN [ ADJ ] .;
:: SOVIET [ ADJ NATION ] .;
:: SOYUZ [ N SATELLITE ] .;
:: SOYUZ-22 [ N SATELLITE ] .;
:: SOYUZ-28 [ N SATELLITE ] .;
:: SOYUZ-TYPE [ ADJ SATELLITE ] .;
:: SP265 [ N SUBNUM ] .;
:: SPACE [ N LOC ] .;
:: SPACECRAFT [ N SCRAFT ] .;
:: SPACEFLIGHT [ N ] .;
:: SPORES [ N ] .;
:: SR-71 [ N TJPE ACRAFT ] .;
:: SS-11 [ N MISSILE ] .;
:: STAGING [ VB PRESP STAGE ] .;
:: STRATEGIC [ ADJ ] .;
:: STRIKE [ N NOMZ ] .;
:: STRIKES [ N NOMZ ] .;
:: SUAM [ N LOC ] .;
:: SUBMARINE [ N ] .;
:: SUBORDINATE [ ADJ ] .;
:: SUCCESSFULLY [ ADVB EVAL ] .;
:: SUDAN [ N LOC ] .;
:: SUDANESE [ ADJ ] .;
:: SUPPORT [ N NOMZ ] .;
:: SURFACE-TO-AIR [ ADJ ] .;
:: SURFACE [ N LOC ] .;
:: SURGUT [ N LOC ] .;
:: SURVEILLANCE [ N NOMZ ] .;
:: SYRIAN [ ADJ NATION ] .;
:: TAIPEI [ N LOC ] .;
:: TAIWAN [ N LOC NATION ] .;
:: TASK [ N NOMZ ] .;
:: TEN [ NUM ] .;
:: TESTING [ VB TRANS PRESP ] .;
:: THAT [ CONJ ] [ RELPRO ] [ ART REF ] .;
:: THE [ ART ] .;
:: THESE [ ART REF ] .;
:: THEY [ PRO ] .;
:: THEIR [ ART POSPRO ] .;
:: THIRD [ ORD ] .;
:: THIS [ ART REF ] .;
:: THOSE [ ART REF ] .;
:: THREE [ NUM ] .;
:: TIME [ N TYME ] .;
:: TO [ PREP EMOD ] .;
:: TOBRUK [ N LOC ] .;
:: TODAY [ N REF TYME ] .;
:: TORORO [ N LOC ] .;
:: TORTOISES [ N ] .;
:: TRAINING [ N ] .;
:: TURNED [ VB PASTP DIR ] [ VB TENSED DIR ] .;
:: TURNING [ VB PRESP DIR ] .;
:: TU-95 [ N TJPE ACRAFT ] .;
:: TWO [ NUM ] .;
:: TYRE [ N LOC ] .;
:: TYURATAM [ N LOC ] .;
:: U-43 [ N TJPE ACRAFT ] .;
:: U1009B [ N SUBNUM ] .;
:: U1211B [ N SUBNUM ] .;
:: U1232 [ N SUBNUM ] .;
:: U1324B [ N SUBNUM ] .;
:: UABC [ N ] .;
:: UAF [ N ] .;
:: UBBC [ N ] .;
:: UG254 [ N SUBNUM ] .;
:: UG836C [ N SUBNUM ] .;
:: UGANDA [ N LOC NATION ] .;
:: UGANDAN [ ADJ NATION ] .;
:: UNDERWAY [ ADJ FLIGHT ] .;
:: UNDETERMINED [ ADJ ] .;
:: UNIDENTIFIED [ ADJ ] .;
:: UNITS [ N PL ] .;
:: VARIOUS [ ADJ ] .;
:: VICINITY [ N ] .;
:: VIOLATED [ VB TRANS PASTP PENETRATE ] [ VB TRANS TENSED PENETRATE ] .;
:: WAS [ BE COPULA TENSED ] .;
:: WEATHER [ N ] .;
:: WERE [ BE COPULA TENSED ] .;
:: WEST [ ADVB DIR ] [ ADJ ] .;
:: WESTERN [ ADJ ] .;
:: WESTWARD [ ADVB DIR ] .;
:: WHICH [ RELPRO PRO ] .;
:: WOULD [ MODAL ] .;
:: X [ N LOC ] .;
:: XB442 [ N SUBNUM ] .;
:: XB262 [ N SUBNUM ] .;
:: ZEILA [ N LOC ] .;
ENDLEX
```

Appendix D - MATRES II Grammar

```
(. SGRAM.4TH -- TEST GRAMMAR FOR SENTENCES)

GRAMMAR DCL/
(. DECLARATIONS)
REGISTER SUBJ
REGISTER OBJ
REGISTER VGRG
REGISTER PASSIVE
REGISTER VHRG
REGISTER DPRG
REGISTER DPF
REGISTER HNRG
REGISTER NPRG
REGISTER SNPF
REGISTER NRG
REGISTER PPRG
REGISTER PREPRG
REGISTER QPRG
REGISTER ARTRG
REGISTER NUMRG
REGISTER VRG
REGISTER CPRG
REGISTER AGRG
REGISTER MONTH
REGISTER YEAR
REGISTER REL
REGISTER RELF
REGISTER RRF
REGISTER RR
REGISTER COMPL
LIST VMODS
LIST NUMLST
LIST PREMODS
LIST POSTMODS
LIST AUX
LIST ADVBLST
LIST PMODS
LIST DAY
LIST TP
LIST DCL
5 LABEL S
4 LABEL NP
3 LABEL DP
2 LABEL QP
2 LABEL NNODE
3 LABEL PP
1 LABEL P
4 LABEL VG
2 LABEL V
```

```
3 LABEL DATE

(. THE DECLARATIVE NET)
:S DCL/
   :PSH RELF GETR 1 = !! TO S/SUBJ * DCL ADDLIST => DCL/S ,,
   :PSH RRF GETR 1 = !!
      VGRG SENDR PASSIVE SENDR VHRG SENDR VMODS SENDL
      TO S/VG * DCL ADDLIST => DCL/S ,,
   :PSH RELF GETR RRF GETR + 0 = !!
      TO S/ * DCL ADDLIST => DCL/S ,,
;;
:S DCL/S
   :CAT [ SCONJ ] !! * DCL ADDLIST => DCL/CONJ ,,
   :JUMP DCL/DCL ,,
;;
:S DCL/CONJ
   :PSH * [ PRESP ] !! TO S/SUBJ * DCL ADDLIST => DCL/S ,,
   :PSH TO NP/ * DCL ADDLIST => DCL/S ,,
;;
:S DCL/DCL
   :POP DCL ,,
;;


(. THE SENTENCE NET)
:S S/
   :PSH TO PP/ * VMODS ADDLIST => S/ ,,
   :PSH TO D/ 0 0 * PP NODE VMODS ADDLIST => S/ ,,
   :CAT [ ADVB TYME ] !! * VMODS ADDLIST => S/PP ,,
   :JUMP S/PP ,,
;;
:S S/PP
   :PSH TO NP/ * SUBJ SETR => S/SUBJ ,,
;;
:S S/SUBJ
   :PSH VMODS SENDL TO VG/
     * VGRG SETR PASSIVE RETR VHRG RETR VMODS RETL => S/VG ,,
;;
:S S/VG
   :PSH PASSIVE GETR 1 = * " BY" AND !! TO AG/
      SUBJ GETR OBJ SETR * SUBJ SETR => S/OBJ ,,
   :JUMP S/OBJ PASSIVE GETR 1 = !!
      SUBJ GETR OBJ SETR 0 SUBJ SETR ,,
   :PSH PASSIVE GETR 0 = VHRG GETR [ TRANS ] AND !!
      TO SNP/ * OBJ SETR => S/OBJ ,,
   :JUMP S/OBJ PASSIVE GETR 0 = !! ,,
;;
:S S/OBJ
   :PSH VMODS SENDL TO VM/ VMODS RETL => S/S ,,
;;
```

```
:S S/S
   :POP SUBJ GETR VGRG GETR OBJ GETR COMPL GETR VMODS S NODE ,,
;;


(. THE NOUN PHRASE SUBNET)
:S NP/
   :PSH TO SNP/ DPRG RETR PREMODS RETL HNRG RETR => NP/SNP ,,
;;
:S NP/SNP
   :PSH TO POSTMODS/ POSTMODS RETL => NP/NP ,,
;;
:S NP/NP
   :POP DPRG GETR PREMODS HNRG GETR POSTMODS NP NODE ,,
;;


(. THE SIMPLE NOUN PHRASE SUBNET)
:S SNP/
   :PSH TO DP/ * DPRG SETR PREMODS RETL 1 SNPF SETR => SNP/DP ,,
   :JUMP SNP/DP ,,
;;
:S SNP/DP
   :PSH PREMODS SENDL TO HN/ * HNRG SETR PREMODS RETL 1 SNPF SETR => SNP/SNP
   :JUMP SNP/SNP ,,
;;
:S SNP/SNP
   :POP SNPF GETR 1 = !!
      DPRG GETR PREMODS HNRG GETR POSTMODS NP NODE ,,
;;


(. THE DETERMINER PHRASE SUBNET)
:S DP/
   :PSH TO QP/ * QPRG SETR 1 DPF SETR => DP/QP ,,
   :JUMP DP/QP ,,
;;
:S DP/QP
   :CAT [ ART ] !! * ARTRG SETR 1 DPF SETR => DP/ART ,,
   :JUMP DP/ART ,,
;;
:S DP/ART
   :CAT [ ADJ ] !! * PREMODS ADDLIST => DP/ART ,,
   :JUMP DP/PREMODS ,,
;;
:S DP/PREMODS
   :PSH TO 1/NUM NUMLST RETL 1 DPF SETR => DP/DP ,,
   :JUMP DP/DP ,,
;;
:S DP/DP
```

```
   :POP DPF GETR 1 = !!
      QPRG GETR ARTRG GETR NUMLST DP NODE ,,
;;


(. THE QUANTIFIER PHRASE SUBNET)
:S QP/
   :CAT [ QUANT ] !! * 0 QP NODE QPRG SETR => QP/QP ,,
   :PSH TO 1/NUM * NUMRG SETR => QP/NUM ,,
;;
:S QP/NUM
   :WRD " OF" !! NUMRG GETR * QP NODE QPRG SETR => QP/QP ,,
;;
:S QP/QP
   :POP QPRG GETR ,,
;;


(. THE NUMBER SUBNET)
:S 1/NUM
   :CAT [ NUMMOD ] !! * NUMLST ADDLIST => 1/NUM ,,
   :CAT [ ADVB ] !! * NUMLST ADDLIST => 1/NUM ,,
   :CAT [ NUM ] !! * NUMLST ADDLIST => 2/NUM ,,
;;
:S 2/NUM
   :JUMP 1/NUM * [ NUMMOD ] * [ ADVB ] * [ NUM ] OR OR !! ,,
   :WRD " AND" !! * NUMLST ADDLIST => 1/NUM ,,
   :WRD " TO" !! * NUMLST ADDLIST => 1/NUM ,,
   :POP NUMLST ,,
;;


(. THE HEAD NOUN SUBNET)
:S HN/
   :CAT [ ADJ ] !! * PREMODS ADDLIST => HN/ ,,
   :CAT [ PRESP ] !! * PREMODS ADDLIST => HN/ ,,
   :CAT [ PASTP ] !! * PREMODS ADDLIST => HN/ ,,
   :PSH * [ N ] !! TO N/ * HNRG SETR => HN/N ,,
;;
:S HN/N
   :JUMP HN/ * [ ADJ ] * [ PRESP ] * [ PASTP ] * [ N ] OR OR OR !!
      HNRG GETR PREMODS ADDLIST ,,
   :POP HNRG GETR ,,
;;


(. THE NOUN SUBNET)
:S N/
   :CAT [ N ] !! * NRG SETR => N/N ,,
;;
```

```
:S N/N
   :PSH * " OF" !! TO PP/ * PPRG SETR => N/PP ,,
   :JUMP N/PP ,,
;;
:S N/PP
   :POP NRG GETR PPRG GETR NNODE NODE ,,
;;


(. THE PREP PHRASE SUBNET)
:S PP/
   :PSH TO 1/NUM * PMODS ADDLIST => PP/NUM ,,
   :CAT [ ADVB ] * [ DIR ] NOT !! * PMODS ADDLIST => PP/UNIT ,,
   :JUMP PP/UNIT ,,
;;
:S PP/NUM
   :CAT [ UNIT ] !! * PMODS ADDLIST => PP/UNIT ,,
;;
:S PP/UNIT
   :CAT [ PREP ] !! * PREPRG SETR => PP/PREP ,,
   :JUMP PP/PREP *-1 " ENROUTE" !! ,,
;;
:S PP/PREP
   :PSH PREPRG GETR [ TYME ] !! TO TP/ * OBJ SETR => PP/PP ,,
   :PSH TO D/ * OBJ SETR => PP/PP ,,
   :PSH TO SNP/ * OBJ SETR => PP/PP ,,
;;
:S PP/PP
   :POP PMODS PREPRG GETR OBJ GETR PP NODE ,,
;;


(. THE POST-HEAD MODIFIER SUBNET)
:S POSTMODS/
   :PSH * [ RELPRO ] !! TO R/ * POSTMODS ADDLIST => POSTMODS/P ,,
   :PSH * [ PRESP ] * [ PASTP TRANS ] OR * [ ADJ ] OR
      * [ ADVB ] *+1 [ PRESP ] *+1 [ PASTP TRANS ] OR *+1 [ ADJ ] OR AND OR !!
      TO RR/ * POSTMODS ADDLIST => POSTMODS/P ,,
   :PSH TO PP/ * POSTMODS ADDLIST => POSTMODS/ ,,
   :JUMP POSTMODS/P ,,
;;
:S POSTMODS/P
   :POP POSTMODS ,,
;;


(. THE RELATIVE CLAUSE SUBNET)
:S R/
   :CAT [ RELPRO ] !! => R/PRO ,,
;;
```

```
:S R/PRO
   :PSH 1 RELF SETR RELF SENDR TO DCL/
      * REL SETR => R/R ,,
;;
:S R/R
   :POP REL GETR ,,
;;


(. THE REDUCED RELATIVE CLAUSE SUBNET)
:S RR/
   :PSH TO RVG/
      * VGRG SETR PASSIVE RETR VHRG RETR VMODS RETL
      => RR/VG ,,
;;
:S RR/VG
   :PSH 1 RRF SETR RRF SENDR VGRG SENDR PASSIVE SENDR VHRG SENDR VMODS SENDL
      TO DCL/ * RR SETR => RR/RR ,,
;;
:S RR/RR
   :POP RR GETR ,,
;;


(. VERB GROUP SUBNET FOR REDUCED RELATIVE CLAUSES)
:S RVG/
   :CAT [ ADVB EVAL ] !! * ADVBLST ADDLIST => RVG/ADVB ,,
   :CAT [ ADVB ] !! * VMODS ADDLIST => RVG/ADVB ,,
   :JUMP RVG/ADVB ,,
;;
:S RVG/ADVB
   :CAT [ PRESP ] !! * VHRG SETR => RVG/VH ,,
   :CAT [ ADJ ] * [ PASTP ] *+1 " BY" AND NOT !!
      * VHRG SETR => RVG/VH ,,
   :CAT [ PASTP ] !! * VHRG SETR 1 PASSIVE SETR => RVG/VH ,,
;;
:S RVG/VH
   :POP ADVBLST AUX 0 VHRG GETR VG NODE ,,
;;


(. THE VERB GROUP SUBNET)
:S VG/
   :PSH VMODS SENDL TO AU/
      AUX RETL ADVBLST RETL VMODS RETL => VG/AUX ,,
;;
:S VG/AUX
   :CAT [ COPULA ] !! * CPRG SETR => VG/COP ,,
   :CAT [ BE ] !! => VG/BE ,,
   :CAT [ VB ] !! * VHRG SETR => VG/VH ,,
;;
:S VG/COP
   :CAT [ ADVB EVAL ] !! * ADVBLST ADDLIST => VG/COP ,,
   :CAT [ ADVB ] !! * VMODS ADDLIST => VG/COP ,,
   :CAT [ ADJ ] CPRG GETR [ BE ] * [ PASTP ] *+1 " BY" AND AND NOT !!
      * VHRG SETR => VG/VH ,,
;;
:S VG/BE
   :CAT [ ADVB EVAL ] !! * ADVBLST ADDLIST => VG/BE ,,
   :CAT [ ADVB ] !! VMODS ADDLIST => VG/BE ,,
   :CAT [ PASTP ] !! * VHRG SETR 1 PASSIVE SETR => VG/VH ,,
;;
:S VG/VH
   :POP ADVBLST AUX CPRG GETR VHRG GETR VG NODE ,,
;;


(. THE AUXILIARY SUBNET)
:S AU/
   :CAT [ ADVB EVAL ] !! * ADVBLST ADDLIST => AU/ ,,
   :CAT [ ADVB ] !! * VMODS ADDLIST => AU/ ,,
   :CAT [ MODAL ] !! * AUX ADDLIST => AU/MDL ,,
   :JUMP AU/MDL ,,
;;
:S AU/MDL
   :CAT [ ADVB EVAL ] !! * ADVBLST ADDLIST => AU/MDL ,,
   :CAT [ ADVB ] !! * VMODS ADDLIST => AU/ ,,
   :CAT [ HAVE ] !! * AUX ADDLIST => AU/HAVE ,,
   :JUMP AU/PASTP ,,
;;
:S AU/HAVE
   :CAT [ ADVB EVAL ] !! * ADVBLST ADDLIST => AU/HAVE ,,
   :CAT [ ADVB ] !! * VMODS ADDLIST => AU/HAVE ,,
   :JUMP AU/PASTP * [ PASTP ] !! ,,
;;
:S AU/PASTP
   :CAT [ BE ] !! * AUX ADDLIST => AU/BE ,,
   :JUMP AU/AU ,,
;;
:S AU/BE
   :CAT [ ADVB EVAL ] !! * ADVBLST ADDLIST => AU/BE ,,
   :CAT [ ADVB ] !! * VMODS ADDLIST => AU/BE ,,
   :JUMP AU/AU * [ PRESP ] !! ,,
;;
:S AU/AU
   :POP AUX ,,
;;


(. THE AGENT NET)
```

```
:S AG/
   :WRD " BY" !! => AG/BY ,,
;;
:S AG/BY
   :PSH TO NP/ * AGRG SETR => AG/AG ,,
;;
:S AG/AG
   :POP AGRG GETR ,,
;;


(. THE VMODS SUBNET)
:S VM/
   :JUMP VM/VM * [ SCONJ ] !! ,,
   :PSH *-1 [ N ] * [ RELPRO ] AND !! TO R/ * VMODS ADDLIST => VM/VM ,,
   :PSH * [ PRESP ] * [ PASTP TRANS ] OR * [ ADJ ] OR
      * [ ADVB ] *+1 [ PRESP ] *+1 [ PASTP TRANS ] OR *+1 [ ADJ ] OR AND
      OR *-1 [ N ] AND !!
      TO RR/ * VMODS ADDLIST => VM/VM ,,
   :PSH TO PP/ * VMODS ADDLIST => VM/ ,,
   :CAT [ ADVB DIR ] !! * VMODS ADDLIST => VM/ ,,
   :PSH TO D/ 0 0 * PP NODE VMODS ADDLIST => VM/ ,,
   :JUMP VM/VM ,,
;;
:S VM/VM
   :POP VMODS ,,
;;


(. THE TIME PHRASE SUBNET)
:S TP/
   :WRD " THE" !! * TP ADDLIST => TP/THE ,,
   :CAT [ ADVB EVAL ] !! * TP ADDLIST => TP/ADVB ,,
   :JUMP TP/ADVB ,,
;;
:S TP/THE
   :CAT [ ADJ ] !! * TP ADDLIST => TP/ADJ ,,
   :JUMP TP/ADJ ,,
;;
:S TP/ADJ
   :CAT [ N TYME ] !! * TP ADDLIST => TP/T ,,
   :CAT [ 4DIG ] !! * TP ADDLIST => TP/T ,,
;;
:S TP/T
   :CAT [ TYME UNIT ] !! * TP ADDLIST => TP/TP ,,
;;
:S TP/ADVB
   :CAT [ N TYME ] !! * TP ADDLIST => TP/1T ,,
   :CAT [ 4DIG ] !! * TP ADDLIST => TP/1T ,,
;;
```

```
:S TP/1T
   :WRD " AND" !! * TP ADDLIST => TP/CONJ ,,
   :WRD " TO" !! * TP ADDLIST => TP/CONJ ,,
   :JUMP TP/CONJ ,,
   :JUMP TP/TP ,,
;;
:S TP/CONJ
   :CAT [ N TYME ] !! * TP ADDLIST => TP/TP ,,
   :CAT [ 4DIG ] !! * TP ADDLIST => TP/TP ,,
;;
:S TP/TP
   :POP TP ,,
;;


(. THE DATE SUBNET)
:S D/
   :CAT [ 1DIG ] !! * DAY ADDLIST => D/DAY1 ,,
   :CAT [ 2DIG ] !! * DAY ADDLIST => D/DAY1 ,,
   :WRD " THE" !! * DAY ADDLIST => D/ART ,,
   :JUMP D/ART ,,
;;
:S D/DAY1
   :CAT [ 1DIG ] !! * DAY ADDLIST => D/DAY ,,
   :CAT [ 2DIG ] !! * DAY ADDLIST => D/DAY ,,
   :JUMP D/DAY ,,
;;
:S D/ART
   :CAT [ ORD ] !! * DAY ADDLIST => D/DAY ,,
;;
:S D/DAY
   :CAT [ MO ] !! * MONTH SETR => D/MO ,,
;;
:S D/MO
   :CAT [ 2DIG ] !! * YEAR SETR => D/D ,,
   :CAT [ 4DIG ] !! * YEAR SETR => D/D ,,
   :JUMP D/D ,,
;;
:S D/D
   :POP DAY MONTH GETR YEAR GETR DATE NODE ,,
;;
ENDGRAMMAR
```

Appendix E - Test Corpus

>> SIX AUXILIARY AVIATION B-60 BUFF HEAVY BOMBERS FROM EABC RGT S1234B AT MOGADISHU CONDUCTED FLIGHTS OVER THE ARABIAN SEA BETWEEN 0220 AND 0634Z ON 21 FEBRUARY.

>> THE PROBABLE FOUR EAFAF B-60 AIRCRAFT FROM XB442 WHICH CONDUCTED OPERATIONS OVER THE NORTH INDIAN OCEAN ARE CURRENTLY ENROUTE HOMEBASE.

>> EIGHT ETHIOPIAN AUXILIARY AVIATION NORTHWEST BOMBER CORPS B-60 BUFF HEAVY BOMBERS FROM REGIMENT E1651D AT MASSAWA ARE CURRENTLY ACTIVE OVER THE GULF OF ADEN.

>> TWO B-60 BUFF A AIRCRAFT FROM THE EAFAF NATIONAL GUARD AUXILIARY SURVEILLANCE AIR REGIMENT XB442 AT ZEILA, ARE CURRENTLY ACTIVE OVER THE NORTH INDIAN OCEAN EAST OF KENYA.

>> EIGHT UBBC B-60 FROM ENTEBBE, STAGING FROM GULU, CONDUCTED FLIGHT OPERATIONS ALONG THE SUDANESE COAST.

>> AT LEAST TEN UGANDAN AUXILIARY AVIATION HEAVY BOMBERS ARE CURRENTLY ENROUTE TO THE RED SEA.

>> TWO INDIAN OCEAN FLEET AIR FORCE B-60 BUFF A AIRCRAFT SUBORDINATE TO NATIONAL GUARDS AUXILIARY SURVEILLANCE AIR REGIMENT AT NAIROBI ARE CURRENTLY ACTIVE OVER THE NORTHERN INDIAN OCEAN IN AN AREA NORTHEAST OF THE SEYCHELLES.

>> TWO EAFAF B-60 BUFF C AIRCRAFT SUBORDINATE TO THE NATIONAL SURVEILLANCE AIR REGIMENT XB262 AT NAIROBI CONDUCTED A PROBABLE RECONNAISSANCE OF A SOUTH AFRICAN TASK GROUP SOUTHEAST OF MALAGASY.

>> TEN UGANDAN CONTINENTAL AVIATION UBBC U1211B ETBF B-60'S DEPLOYED TO GULU.

>> THREE BC254 B-61 BUFF O AIRCRAFT CONDUCTED AN OPEN OCEAN MARITIME RECONNAISSANCE NAVIGATIONAL TRAINING MISSION TO THE GENERAL AREA SOUTHEAST OF THE LACCADIVES ON 21/22 FEBRUARY.

>> THE THREE BC254 BUFF D AIRCRAFT WHICH CONDUCTED AN OPEN OCEAN MARITIME RECONNAISSANCE NAVIGATIONAL TRAINING MISSION TO THE GENERAL AREA SOUTHEAST OF THE LACCADIVE ISLANDS, HAVE RETURNED TO UGANDA BY 0620Z.

>> TWO B-60 BUFF C AIRCRAFT FROM NATIONAL SURVEILLANCE AIR REGT UG254 AT GULU, ARE CURRENTLY OPERATING IN THE AREA OF THE NYANJA TASK GROUP JUST EAST OF THE SEYCHELLES.

>> TWO EAFAF MOMBASA-BASED KE843 B-60 BUFF C AIRCRAFT ARE POSSIBLY CONDUCTING COMMAND AND CONTROL OPERATIONS OVER THE ARABIAN SEA POSSIBLY IN CONJUNCTION WITH A RETURNING DELTA-CLASS SUBMARINE.

>> THE TWO EAFAF MOMBASA-BASED KE843 B-60 BUFF C AIRCRAFT WHICH WERE POSSIBLY CONDUCTING COMMAND AND CONTROL OPERATIONS OVER THE SOUTHERN ARABIAN SEA, HAVE RETURNED TO THE AFRICAN LANDMASS BY 1030Z.

>> TWO SOUTH AFRICAN SAFAF CAPETOWN-BASED SA554 B-60 BUFF D ARE PRESENTLY LOCATED OVER THE SOUTH ATLANTIC, POSSIBLY IN REACTION TO TWO NIGERIAN UNITS LOCATED EAST OF ST HELENA.
>> THIS FLIGHT ACTIVITY WAS PRECEDED BY A WEATHER RECONNAISSANCE FLIGHT BY ONE PRETORIA-BASED SP265 B-80 BEACON TO THE CAPE VERDE ISLANDS.

>> THE SIX SOUTH AFRICAN FLEET AIR FORCE SR-71 ACFT WHICH CONDUCTED ASW ASSOCIATED OPERATIONS OVER THE ARABIAN SEA DURING THE PREVIOUS PERIOD HAVE RETURNED TO NORMAL OPERATING AREAS.

>> THE MISSION DESTINATION OF THESE AIRCRAFT IS UNDETERMINED AT THIS TIME.

>> AIRCRAFT CONDUCTED OPERATIONS IN AN AREA SOUTH OF MALAGASY.

>> AIRCRAFT ARE ACTIVE OVER THE NORTHERN INDIAN OCEAN, NORTH OF THE SEYCHELLE ISLANDS.
>> AIRCRAFT ARE CURRENTLY CONDUCTING UNDETERMINED OPERATIONS IN AN AREA SOUTHEAST OF THE MALDIVE ISLANDS.

>> AIRCRAFT CONDUCTED FLIGHT OPERATIONS OVER THE ARABIAN SEA BETWEEN 1035 AND 2219Z.

>> TWO POSSIBLY FOUR SAFAF CAPETOWN-BASED SA622 BUFF D AIRCRAFT, STAGING FROM PRETORIA, ARE CURRENTLY OPERATING OVER THE ARABIAN SEA.

>> TWO ARE CURRENTLY ENROUTE RED SEA.

>> TWO MEDIUM BOMBERS FROM REGIMENT U1211B AT GULU 3247N3218E CONDUCTED A RECONNAISSANCE FLIGHT OVER THE COMBAT AREA BETWEEN 1146 AND 1432Z.
>> ACTY WAS PRIMARILY ASW IN NATURE.

>> TWO NAIROBI-BASED SR-71 FODDER AIRCRAFT ARE CURRENTLY ENROUTE HOMEBASE AFTER CONDUCTING AN INTELLIGENCE COLLECTION FLIGHT ALONG THE SEYCHELLE ISLAND CHAIN.

>> TWO NAIROBI-BASED SR-71 FODDER AIRCRAFT ARRIVED IN THE VICINITY OF HOMEBASE AFTER CONDUCTING AN INTELLIGENCE COLLECTION FLIGHT ALONG THE SEYCHELLE ISLAND CHAIN BETWEEN 0311-0401Z.

>> AT LEAST 46, POSSIBLY 65, SAFAF B-60 BUFF AIRCRAFT REPRESENTING THE THREE STRATEGIC REGIMENTS IN THE SAFAF ARE PRESENTLY LOCATED OVER THE ARABIAN SEA IN A PROBABLE SIMULATED AIR-TO-SURFACE EXERCISE INVOLVING VARIOUS SOUTH AFRICAN SURFACE UNITS.

>> TWO SAFAF CAPETOWN-BASED SC462 B-60 BUFF C AIRCRAFT ARE CURRENTLY OPERATING OVER THE ARABIAN SEA.

>> ALL ACFT PROBABLY RETURNED TO NORMAL OPERATING AREAS BY 1112Z.

>> TWO SAFAF CAPETOWN-BASED SC462 B-60 BUFF C AIRCRAFT OPERATED OVER THE ARABIAN SEA IN PROBABLE RECONNAISSANCE SUPPORT FOR THE STRIKE ACFT.

>> AT LEAST ONE UGANDAN AUXILIARY AVIATION B-60 BUFF WAS ACTIVE OVER THE INDIAN OCEAN BETWEEN 0538-0920Z ON 11 FEBRUARY.

>> TWO SOUTH AFRICAN FLEET AIR FORCE SAFAF B-60 BUFF AIRCRAFT PRESENTLY DEPLOYED TO MAURITIUS, WERE ACTIVE IN POSSIBLE COMMUNICATION EQUIPMENT TESTING FLIGHT.

>> EIGHT UAF GULU-BASED UG836C B-63 BEACON AIRCRAFT, POSSIBLY STAGING FROM ENTEBBE, CONDUCTED A POSSIBLE SMALL SCALE ADX OVER THE BUJUMBURA COMPLEX DURING THE EARLY 0200Z HOUR.
>> THE AIRCRAFT PROBABLY RETURNED TO NORMAL OPERATING AREA BY THE MID 1100Z HOUR.

>> TWO UGANDAN AUXILIARY AVIATION UBBC B-63 BEACON MEDIUM BOMBERS FROM REGIMENT U1324B AT GULU 0247N3218E DEPLOYED TO ENTEBBE 0003N3227E BETWEEN 0115 AND 0332Z.

>> CURRENTLY, THREE ADDITIONAL U1324B B-63S ARE PROBABLY ENROUTE TO ENTEBBE.

Appendix F - Examples of System Operation

EXAMPLE 1

```
*>> TWO UGANDAN ACFT FROM REGIMENT A313 AT ENTEBBE DEPLOYED TO GULU
*AT 0200Z ON 21 FEBRUARY.
PARSE OUTPUT:
LIST OF:
| NODE: 1|S
| | LIST OF:
| | | NODE: 2|PP
| | | | NODE: 4|DATE
| | | | | <<NIL>>
| | | | | 392.. FEBRUARY
| | | | | LIST OF:
| | | | | | 372.. 21
| | | | | END LIST
| | | | END NODE
| | | | 352.. ON
| | | | LIST OF:
| | | | END LIST
| | | END NODE
| | | NODE: 2|PP
| | | | LIST OF:
| | | | | 332.. 0200Z
| | | | END LIST
| | | | 312.. AT
| | | | LIST OF:
| | | | END LIST
| | | END NODE
| | | NODE: 2|PP
| | | | NODE: 2|NP
| | | | | LIST OF:
| | | | | END LIST
| | | | | NODE: 5|NNOD
| | | | | | <<NIL>>
| | | | | | 292..GULU
| | | | | END NODE
| | | | | LIST OF:
| | | | | END LIST
| | | | | <<NIL>>
| | | | END NODE
| | | | 272..TO
| | | | LIST OF:
| | | | END LIST
| | | END NODE
| | END LIST
| | <<NIL>>
| | <<NIL>>
| | NODE: 2|VG
| | | 232.. DEPLOYED
| | | <<NIL>>
| | | LIST OF:
| | | END LIST
| | | LIST OF:
| | | END LIST
| | END NODE
| | NODE: 2|NP
| | |  LIST OF:
| | | | NODE: 2|PP
| | | | | NODE: 2|NP
| | | | | | LIST OF:
| | | | | | END LIST
| | | | | | NODE: 5|NNOD
| | | | | | | <<NIL>>
| | | | | | | 212.. ENTEBBE
| | | | | | END NODE
| | | | | | LIST OF:
| | | | | | END LIST
| | | | | | <<NIL>>
| | | | | END NODE
| | | | | 192.. AT
| | | | | LIST OF:
| | | | | END LIST
| | | | END NODE
| | | | NODE: 2|PP
| | | | | NODE: 2|NP
| | | | | | LIST OF:
| | | | | | END LIST
| | | | | | NODE: 5|NNOD
| | | | | | | <<NIL>>
| | | | | | | 172..A313
| | | | | | END NODE
| | | | | | LIST OF:
| | | | | | | NODE: 5|NNOD
| | | | | | | | <<NIL>>
| | | | | | | | 182.. REGIMENT
| | | | | | | END NODE
| | | | | | END LIST
| | | | | | <<NIL>>
| | | | | END NODE
| | | | | 152.. FROM
| | | | | LIST OF:
| | | | | END LIST
| | | | END NODE
| | | END LIST
| | | NODE: 5|NNOD
| | | | <<NIL>>
| | | | 132..ACFT
| | | END NODE
| | | LIST OF:
| | | | 112.. UGANDAN
| | | END LIST
| | | NODE: 2|DP
| | | | LIST OF:
| | | | | 92.. TWO
| | | | END LIST
| | | | <<NIL>>
| | | | <<NIL>>
| | | END NODE
| | END NODE
| END NODE
END LIST
Event: DEPLOY
Object:
...Equipment= UGANDAN ACFT
...Nationality= UGANDAN
...Subordination= FROM REGIMENT A313
...Stagingbase= AT ENTEBBE
...Number= TWO
Destination= TO GULU
Time= AT 0200Z
Date= ON 21 FEBRUARY
EVENT RECORD COMPLETE.
```

EXAMPLE 2

```
*>> THE ACFT WERE ENROUTE TO NAIROBI BETWEEN 0200Z AND 0400Z
*   ON 21 FEBRUARY 1965.
PARSE OUTPUT:
LIST OF:
| NODE: 1|S
| | LIST OF:
| | | NODE: 2|PP
| | | | NODE: 4|DATE
| | | | | 350.. 1965
| | | | | 330.. FEBRUARY
| | | | | LIST OF:
| | | | | | 310.. 21
| | | | | END LIST
| | | | END NODE
| | | | 290.. ON
| | | | LIST OF:
| | | | END LIST
| | | END NODE
| | | NODE: 2|PP
| | | | LIST OF:
| | | | | 270.. 0400Z
| | | | | 250.. AND
| | | | | 230.. 0200Z
| | | | END LIST
| | | | 210.. BETWEEN
| | | | LIST OF:
| | | | END LIST
| | | END NODE
| | | NODE: 2|PP
| | | | NODE: 2|NP
| | | | | LIST OF:
| | | | | END LIST
| | | | | NODE: 5|NNOD
| | | | | | <<NIL>>
| | | | | | 190.. NAIROBI
| | | | | END NODE
| | | | | LIST OF:
| | | | | END LIST
| | | | | <<NIL>>
| | | | END NODE
| | | | 170.. TO
| | | | LIST OF:
| | | | END LIST
| | | END NODE
| | END LIST
| | <<NIL>>
| | <<NIL>>
| | NODE: 2|VG
| | | 150.. ENROUTE
| | | 130.. WERE
| | | LIST OF:
| | | END LIST
| | | LIST OF:
| | | END LIST
| | END NODE
| | NODE: 2|NP
| | | LIST OF:
| | | END LIST
| | | NODE: 5|NNOD
| | | | <<NIL>>
| | | | 110.. ACFT
| | | END NODE
| | | LIST OF:
| | | END LIST
| | | NODE: 2|DP
| | | | LIST OF:
| | | | END LIST
| | | | 90.. THE
| | | | <<NIL>>
| | | END NODE
| | END NODE
| END NODE
END LIST
Event: ENROUTE
Object:
...Equipment= ACFT
...Number=
Destination= TO NAIROBI
Time= BETWEEN 0200Z AND 0400Z
Date= ON 21 FEBRUARY 1965
EVENT RECORD COMPLETE.
```

EXAMPLE 3

```
*>> THE TWO ACFT WERE ENROUTE TO NAIROBI ON RECONNAISSANCE.
PARSE OUTPUT:
LIST OF:
| NODE: 1|S
| | LIST OF:
| | | NODE: 2|PP
| | | | NODE: 2|NP
| | | | | LIST OF:
| | | | | END LIST
| | | | | NODE: 5|NNOD
| | | | | | <<NIL>>
| | | | | | 228.. RECONNAISSANCE
| | | | | END NODE
| | | | | LIST OF:
| | | | | END LIST
| | | | | <<NIL>>
| | | | END NODE
| | | | 208.. ON
| | | | LIST OF:
| | | | END LIST
| | | END NODE
| | | NODE: 2|PP
| | | | NODE: 2|NP
| | | | | LIST OF:
| | | | | END LIST
| | | | | NODE: 5|NNOD
| | | | | | <<NIL>>
| | | | | | 188.. NAIROBI
| | | | | END NODE
| | | | | LIST OF:
| | | | | END LIST
| | | | | <<NIL>>
| | | | END NODE
| | | | 168.. TO
| | | | LIST OF:
| | | | END LIST
| | | END NODE
| | END LIST
| | <<NIL>>
| | <<NIL>>
| | NODE: 2 |VG
| | | 148.. ENROUTE
| | | 128.. WERE
| | | LIST OF:
| | | END LIST
| | | LIST OF:
| | | END LIST
| | END NODE
| | NODE: 2|NP
| | | LIST OF:
| | | END LIST
| | | NODE: 5|NNOD
| | | | <<NIL>>
| | | | 108.. ACFT
| | | END NODE
| | | LIST OF:
| | | END LIST
| | | NODE: 2|DP
| | | | LIST OF:
| | | | | 88.. TWO
| | | | END LIST
| | | | 68.. THE
| | | | <<NIL>>
| | | END NODE
| | END NODE
| END NODE
END LIST
Event: ENROUTE
Object:
...Equipment= ACFT
...Number= TWO
Mission= ON RECONNAISSANCE
Destination= TO NAIROBI
EVENT RECORD COMPLETE.
```

Appendix G - MATRES II Operations

1.0 Introduction

These instructions apply only to the OSI PDP-11/45 running under the standard OSI RSX-11D system. On that system, the MATRES II files are stored under two UFDs: [60,4] contains all programs and data except for the template file; this file and the ERL compiler are stored under [60,5].

2.0 Compiling the ERL Templates

The source for the templates is in the file [60,5]TEMPLATE.ERL. This file, like all MATRES II files, may be modified with any standard editor. The compilation procedure is as follows (assuming that you are logged in under [60,5]):

If the SPTBOL program has not been installed:
```
MCR>INS [11,50]SPTBOL/TASK=...SPT
```
Then,
```
MCR>@ERLCMPL
```

The command file will do the rest. If any syntax errors are discovered, the offending clause(s) will be printed on the console. Two files will be created: TEMPLATE.INT, the output from Pass 1, and TEMPLATE.4TH, the final output, which will be input to MATRES.

3.0 Running the MATRES System

You should be logged in under [60,4]. The procedure is:

If the FORTH program has not been installed:
```
MCR>INS [60,3]FORTH/TASK=...4TH/INC=25500
```
Then,
```
MCR>4TH
```
After Forth says hello,
```
*FORTH LOAD
*@MATRES
```

After "MATRES READY!" is printed, sentences may be entered, preceded by ">> " and terminated by a period. Sentences may span multiple lines, breaking at word boundaries. Example:

```
*>> TWO UGANDAN ACFT FROM A313 AT ENTEBBE DEPLOYED TO
*GULU AT 0200Z ON 21 FEBRUARY.
Event: DEPLOY
Object:
...Equipment= UGANDAN ACFT
...Number= TWO
...Subordination= FROM A313
...Stagingbase= AT ENTEBBE
Destination= TO GULU
Time= AT 0200Z
Date= ON 21 FEBRUARY
```

There are various debugging switches available to enable printout of intermediate results. To set a switch, enter "name 1SET"; to reset it, enter "name 0SET". The following switches exist currently:

DEBUG	causes a printout of the lexical units as they are found by lexical processing, followed by a trace of the parsing process. The state names are shown in Forth format, as a number of characters followed by the first four characters of the name.
PRTREE	causes the parse tree to be printed out after a successful parse. The elements of a node and members of a list will be shown in reverse order from that input to the templates. The node names also are in Forth format.
P_SW	causes a trace of the unification process. For every term in the head of a clause (and every term in a skeleton in the head, etc.), the corresponding goal term is printed.
PTRY	causes a trace of clause entries. Clause names are printed in Forth format, and consist of a "C" followed by a number. This will be the number of the clause in the original source file, starting from zero. For instance, "C27" will be the 27th clause from the top one in the TEMPLATE.ERL file.

END